Across Token and Token Distributor Audit

OPENZEPPELIN SECURITY | JULY 21, 2022 Security Audits

May 10th, 2022

This security assessment was prepared by **OpenZeppelin**, protecting the open economy.

Table of Contents

Summary

Type
: DeFi

Timeline
: From 2022-04-25
: To 2022-05-10

Languages
: Solidity

Total Issues
: 13 (13 resolved)

Critical Severity Issues
: 0 (0 resolved)

High Severity Issues
: 1 (1 resolved)

Medium Severity Issues
: 1 (1 resolved)

Low Severity Issues
: 4 (4 resolved)

Notes & Additional Information
: 7 (7 resolved)

Scope

We audited the `across-protocol/across-token` repository at commit `42130387f81debf2a20d2f7b40d9f0ccc1dcd06a`.

In scope were the following contracts:

System Overview

The system is composed of the Across token (ACX) and the Across token distributor. Together, these are meant to facilitate a liquidity mining program that will accompany the launch of the most recent version of the Across protocol.

The Across token is an ERC20 compliant token.

The Across token distributor allows users to stake whitelisted ERC20 tokens (LP tokens) to earn ACX rewards. At deposit time depositors earn a fixed base emission rate. The longer depositors keep their tokens staked, the higher the reward rate they earn. The reward rate is capped at a value set by the owner of the distributor. Sequential deposits result in an average deposit time as a weighted average of previous deposits. If at any point the depositor claims their rewards or unstakes all LP tokens, then their rewards emission rate is reset to the base rate. The contract is designed to hold multiple LP tokens, each with independent parameterization for liquidity mining.

Privileged Roles

Each contract has only one privileged role, which is the contract's respective `owner`.

The Across token's owner can mint and burn the token. The owner can also renounce and transfer ownership.

The Across token distributor's owner can also transfer and renounce ownership. In addition, the owner is the only account which receives tokens recovered from the contract, which are comprised of tokens which were mistakenly sent to and are retrievable from the contract. Finally, the owner can enable and disable tokens for staking and change parameters of enabled LP tokens. These parameters are set per-token and include the base ACX emission rate, an emission rate cap, and time needed to reach the emission rate cap.

Findings

Here we present our findings.

The `recoverErc20` function is meant to facilitate the recovery of any ERC20 tokens that may be mistakenly sent to the `AcceleratingDistributor` contract. As this is a public function with no modifiers, anyone can call this function to transfer an ERC20 token from the `AcceleratingDistributor` contract to the owner of `AcceleratingDistributor`. The only ERC20 tokens that are explicitly disallowed from being recovered are `stakedToken`s that have already been initialized in the system.

However, it is currently possible to recover the ERC20 `rewardToken` using the `recoverErc20` function. Doing so would transfer some specified `amount` of `rewardToken` from the `AcceleratingDistributor` contract to the contract's owner. This would, subsequently, prevent stakers from being able to access their rewards because `AcceleratingDistributor` could be left with an insufficient balance of `rewardToken`s.

Even if the owner were to send `rewardToken`s back to the `AcceleratingDistributor` contract, a malicious actor could immediately transfer all of the `rewardTokens` back to the owner. Redeployment would be necessary to fix the issue.

Consider disallowing recovery of the `rewardToken` within the `recoverErc20` function.

***Update:** Fixed as of commit `bcdabc06ca6d789b95c5b26d23f48dab8bfad277` in pull request #5.*

Medium Severity

Lack of input validation

The codebase generally lacks sufficient input validation.

In the `AcceleratingDistributor` contract, the `enableStaking` function allows the contract owner to configure several parameters associated with a `stakedToken`. Several of these parameters have no input checking. Specifically:

- The `maxMultiplier` parameter has no upper or lower bound.

could cause the `getUserRewardMultiplier` function to revert on overflow. This could, in turn, cause calls to the `getOutstandingRewards` and `_updateReward` functions to revert. This would interfere with the normal operation of the system. However, it could be fixed by the contract owner using the `enableStaking` function to update to a more reasonable `maxMultiplier`.

- Similarly, the `secondsToMaxMultiplier` parameter has no lower bound. If allowed to be zero then `getUserRewardMultiplier` could revert due to division by zero. This could cause the `getOutstandingRewards` and `_updateReward` functions to revert as outlined above. The contract owner could put the system back into a stable state by making `secondsToMaxMultiplier` non-zero.
- The `baseEmissionRate` parameter has no upper bound. If set too high then, as soon as `stakingToken.cumulativeStaked` were some non-zero value, the `baseRewardPerToken` function would always revert due to an overflow. Importantly, this value could be set to a system destabilizing value even when `stakingToken.cumulativeStaked` was already non-zero. This would cause `_updateReward` to revert even in the case that the provided account was the zero address. This detail would prevent the contract owner from fixing the situation without a complete redeployment of the system if any `stakedToken` at all were actively being staked. Any `stakedToken` already in the contract would be locked.

To avoid errors and unexpected system behavior, consider implementing `require` statements to validate *all* user-controlled input.

Update: *Fixed as of commit* `9652a990a14d00e4d47f9e4f3df2c422a6881d4a` *in pull request #6 and commit* `3a4a6382b88f40fa9d816e5e5b1d3a31a7b24f27` *in pull request #20.*

Low Severity

Incomplete and unindexed events

Throughout the codebase, events are used to signify when changes are made to the state of the system. However, all of the existing events lack indexed parameters. Some events are missing

- The `TokenEnabledForStaking`, `RecoverErc20`, `Stake`, `Unstake`, `GetReward`, and `Exit` events in the `AcceleratingDistributor` contract.

Event emissions which do not fully track state changes include:

- The `RecoverErc20` event does not emit the caller, which may be of interest to those interested in calls to the `recoverErc20` function where the event is emitted.
- The `Stake`, `Unstake`, and `Exit` events are emitted when a user calls the `stake`, `unstake`, or `exit` functions, respectively. These functions update the `stakedToken`'s `cumulativeStaked` value, but this is not emitted in the events. Similarly, these functions – as well as the `getReward` function – update a `stakedToken`'s `lastUpdateTime` and `rewardPerTokenStored` as well as a `userDeposit`'s `rewardsOutstanding` and `rewardsPaidPerToken` values, but these state changes are not captured by any associated event emissions.

Consider completely indexing existing events, adding new indexed parameters where they are lacking, and/or adding new events to facilitate off-chain services searching and filtering for events. Consider emitting all events in such a complete manner that they could be used to rebuild the state of the contract.

***Update:** Fixed as of commit `71178ca0ce24633eb4644f64094f455862edf5e9` in pull request #7 and commit `3a4a6382b88f40fa9d816e5e5b1d3a31a7b24f27` in pull request #20.*

Recovery function is not robust

Within the `AcceleratingDistributor` contract, the `recoverErc20` function is intended to facilitate the recovery of tokens which may have been sent to the contract in error. Without such a function, tokens sent to the contract in error would generally become completely inaccessible.

However, the current implementation of the recovery function is less robust than may be desirable.

For instance, all recovered tokens are hard-coded to be sent to the contract's owner. But the contract also has the ability to relinquish ownership, whereby the owner is set to the zero address.

Additionally, any tokens that have ever been enabled for staking are not recoverable. Unfortunately, it is precisely tokens that *are or were* eligible to be staked that are most likely to be sent to the contract in error, especially by users trying to stake with the protocol in an incorrect manner.

To better separate concerns, consider having tokens recovered to a recovery address that is independent of the owner. Alternatively, consider explicitly documenting the fact that token recoverability is dependent on an owner address that can be controlled. Also, since the `cumulativeStaked` amount for every staking token is already accounted for, consider using it to allow for the recovery of staking tokens which were sent in error and are, as a result, unaccounted for.

***Update:** Fixed as of commit `0cc740ecc30694077d44674d80b48c4e6d75ce31` in pull request #8.*

Unhandled failures in token interactions

There are ERC20 `transfer` operations, executed over potentially untrusted ERC20 tokens, which are not correctly wrapped to ensure that they are always safe and behave as expected.

Specifically, for the `transfer` on line 177 of the `AcceleratingDistributor` contract, the current implementation does not handle a case where the call to `transfer` fails by returning a `false` boolean value (rather than reverting the transaction).

For more predictable and consistent behavior, consider using the OpenZeppelin `SafeERC20` library which is used for other transfers throughout the codebase, as it implements wrappers around ERC20 operations that return `false` on failure.

***Update:** Fixed as of commit `3722baff5cf0ee936b43ecf07ae47b44b3f5688d` in pull request #9.*

Unused inherited contracts

The `AcceleratingDistributor` contract imports and inherits from the `Pausable` contract but does not use any of the inherited functionality.

***Update:** Fixed as of commit `b511e0d96a18d1087da60e2e02ee18120eb0a291` in pull request #10.*

Notes & Additional Information

Visibility unnecessarily permissive

The `mint` and `burn` functions of the `AcrossToken` contract and `enableStaking`, `stake`, `getCumulativeStaked`, and `getUserStake` of the `AcceleratingDistributor` contract are marked as `public`, while they could be marked as `external`.

***Update:** Fixed as of commit `0f027e6cb8ae146a755de3041172efc76bb87d5f` in pull request #11 and commit `3a4a6382b88f40fa9d816e5e5b1d3a31a7b24f27` in pull request #20.*

System is incompatible with non-standard ERC20 tokens

Currently, the system is incompatible with non-standard ERC20 tokens and enabling such tokens for staking could lead to loss of funds in the following ways:

1. The system's internal accounting mechanism does not currently support tokens that can charge fees on transfers, such as Tether (USDT) (the most widely known asset with this feature). This means that such tokens' `transfer` and `transferFrom` functions could potentially fail to increase the recipient's balance by the amount that is actually transferred.If `AcceleratingDistributor` allows staking of tokens charging such transfer fees, then stakers would receive more rewards than intended.
2. The `recoverErc20` function is currently callable by any address. This could be problematic if the system were to allow staking of non-standard ERC20 tokens, particularly if those tokens were to have a double entry point. In such a case the current restrictions may not be sufficient to restrict such tokens' recovery via the `recoverErc20` function (e.g. Compound-TUSD Integration Issue Retrospective).

amount of assets deposited. Additionally, consider limiting the [illegible] function so that is can only be called from trusted accounts.

Alternatively, consider documenting that the system is incompatible with non-standard ERC20 tokens and ensuring the contract owner thoroughly vets any ERC20 tokens that are enabled for staking. If the `recoverErc20` function is left unrestricted, then consider moving it out of the `ADMIN` section of the codebase.

***Update:** Fixed as of commit `5ea4099a44e5edfa5e84cc65e744cf546d7f5957` in pull request #12 and commit `3a4a6382b88f40fa9d816e5e5b1d3a31a7b24f27` in pull request #20.*

Misleading function names

In the `AcceleratingDistributor` contract there are multiple functions, storage variables, and events that have misleading names. These could potentially confuse users and lead to unintentional or unexpected consequences when interacting with the contract. For instance:

- The function name `enableStaking` implies that it can only be used to enable a staking token. However, the function can be used to enable or disable a staking token, and/or modify the properties `baseEmissionRate`, `maxMultiplier` or `secondsToMaxMultiplier`. Consider renaming the function to `configureStakingToken`.
- The `enableStaking` function emits the event `TokenEnabledForStaking`. Consider renaming the event to better reflect the actual use case of the emitting function as outlined in the above bullet point.
- The function prefix `get` typically indicates a view function, however the function `getReward` transfers outstanding rewards to the caller. Consider renaming it to `withdrawReward`.
- The function name `getTimeFromLastDeposit` implies that a difference between the current time and the last time of deposit is returned. However, the difference is calculated using the weighted average deposit time. Consider renaming it to `getTimeSinceAverageDeposit`.

the `rewardsOustanding` variable, which contains the funds that are actually withdrawable to the individual user. Consider renaming `rewardsPaidPerToken` to `rewardsAccumulatedPerToken`.

Update: *Fixed as of commit* `5db3215559fad05ebf98ac9f2bd91187a1e442d7` *in pull request #13.*

Missing documentation details

In the `AcceleratingDistributor` contract several parts of the documentation contain missing or misleading details. For instance:

- The functions `getUserRewardMultiplier` and `baseRewardPerToken` return ratios or multipliers represented by a `uint256` value with a fixed precision of 18 decimals. However, the decimal precision used is not documented.
- The contract-level documentation refers to a `maxEmissionRate`. However, the implementation uses a `maxMultiplier` which is multiplied with a `baseEmissionRate` instead.

To increase the overall readability of the codebase and reduce potential confusion, consider clarifying the documentation and adding missing details where appropriate.

Update: *Fixed as of commit* `e3e7cf1c46304accc73f25a70f14036841e81239` *in pull request #14.*

Not using `immutable`

In the `AcceleratingDistributor` contract the `rewardToken` variable could be marked `immutable` given that is only ever set in the constructor.

If `rewardToken` is never meant to be modifiable, then consider marking it `immutable` to better signal intent and reduce gas consumption.

Update: *Fixed as of commit* `973e002a92e646f5e3af74d235a1e77d03bd69a0` *in pull request #15.*

will not simply rehash our previously raised concerns and suggestions, but we do feel it prudent to expound on some of the potential security implications of having test code in production in this particular case:

The `AcceleratingDistributor` contract inherits from the `Testable` contract. This facilitates modification of the `getCurrentTime` function's behavior during testing. This inheritance is meant to be kept in production. This is considered safe because passing the zero address for the parameter `_timer` during deployment will disable the testing module and return `block.timestamp` instead.

The contract logic ensures that all staked tokens can be unstaked by their respective owners and that all rewards can be collected even if further staking is disabled by the contract owner. Calls to `unstake` and `getReward` are guarded by a modifier which ensures only that a token had been enabled for staking at some time – not that it is *currently* enabled for staking. It does this by verifying the token's `lastUpdateTime` is greater than zero. The contract owner is not meant to be able to set `lastUpdateTime` in production and it is guaranteed to be greater than zero for each token that has previously been enabled for staking.

However, a malicious operator could deploy the code with the `address` `_timer` pointing to a smart contract that allows the attacker to arbitrarily change time. They could at first report accurate block times and then wait until a significant portion of staked TVL is accumulated. Later, they could change the reported time to zero. As a consequence, users would be unable to unstake or obtain rewards. What's worse, the malicious operator could then steal all staked TVL via `recoverERC20` (which would no longer recognize the staked token as such when their `lastUpdateTime` was equal to zero).

A potential risk here is that a third party could uses this exact code, misusing the `Testable` functionality to be genuinely malicious. Since ecosystem tools such as Etherscan can do exact matching on bytecode, the malicious deployment could end up being directly linked to the Across protocol, which could lead to some loss of reputation.

As this is a fairly general-purpose, user-facing system, with an open source license, we can certainly imagine it being reused. The subtleties of the deployment can have drastic implications for the security model of the deployed system. Normalizing such designs can have unintended

Consider better isolating test and production code where possible. When this is not possible, consider bolstering the warnings in the code so that even casual users could better understand the implications if system variables are are not set as intended in production.

***Update:** Fixed as of commit* `144293ec5b81c3367ad58738d7bca273b3e8a9e4` *in pull request #16.*

Typographical errors

The codebase contains the following typographical errors:

In `AcceleratingDistributor.sol`:

- On line 17 and line 89 `pro-rate` should be `pro-rata`.
- On line 126, `after the end of the staking program ends` is ungrammatical.
- On line 202, `exists` should be `exits`.
- On line 203, `callers` should be `caller's`.
- On line 228, `the all information associated` should be `all the information associated`.

Consider correcting these typos to improve the overall readability of the codebase.

***Update:** Fixed as of commit* `7975d8ad19a3766341a115fce83c7752412f837b` *in pull request #17.*

Conclusions

0 critical and 1 high severity issues were found. Some changes were proposed to follow best practices and reduce the potential attack surface.

Appendix

Severity Levels

Critical Severity

High Severity

The issue puts a large number of users' sensitive information at risk, or is reasonably likely to lead to catastrophic impact for client's reputation or serious financial implications for client and users.

Medium Severity

The issue puts a subset of users' sensitive information at risk, would be detrimental for the client's reputation if exploited, or is reasonably likely to lead to moderate financial impact.

Low Severity

The risk is relatively small and could not be exploited on a recurring basis, or is a risk that the client has indicated its low impact in view of the client's business circumstances.

Notes & Additional Information

The risk is relatively small and could not be exploited on a recurring basis, or is a risk that the client has indicated its low impact in view of the client's business circumstances. It may also include non-security-relevant content for purely informational purposes.

Related Posts

Beefy Zap Audit

Linea Bridge Audit

OpenBrush is an open-source smart contract library written in the Rust programming language and the...

Security Audits

Security Audits

Security Audits