AlexGo Audit

Alex-v1

November 2021

By CoinFabrik

Introduction

CoinFabrik was asked to audit the contracts for the AlexGo project. First we will provide a summary of our discoveries and then we will show the details of our findings.

Summary

The contracts audited are from the alex-v1 repository at https://github.com/alexgo-io/alex-v1. The audit is based on the commit `0913c9eb0c8f79b3cd0b843641075ce27d437c96`. Fixes were made and rechecked based on the commit `feb43898de520ca1c55a6d8bc5bcf2d6f7df8ffa`.

Contracts

The audited contracts are:

- `clarity/contracts/equations/weighted-equation.clar`: Library for weighted math functions.
- `clarity/contracts/equations/yield-token-equation.clar`: Library for yield token math functions.
- `clarity/contracts/pool/alex-reserve-pool.clar`: Staking contract.
- `clarity/contracts/pool/collateral-rebalancing-pool.clar`: Lending and borrowing pool contract with pools of variable weights.
- `clarity/contracts/pool/fixed-weight-pool.clar`: Contract with fixed-weight liquidity pools.
- `clarity/contracts/pool/liquidity-bootstrapping-pool.clar`: Contract with liquidity pools that increase bootstrapped token weight over the time.
- `clarity/contracts/pool/yield-token-pool.clar`: Liquidity pools for token and correspondent yield token pairs.

Analyses

The following analyses were performed:

- Misuse of the different call methods
- Integer overflow errors
- Division by zero errors
- Front running attacks
- Reentrancy attacks
- Misuse of block timestamps
- Softlock denial of service attacks
- Functions with excessive gas cost
- Missing or misused function qualifiers
- Needlessly complex code and contract interactions
- Poor or nonexistent error handling
- Failure to use a withdrawal pattern
- Insufficient validation of the input parameters
- Incorrect handling of cryptographic signatures

Findings and Fixes

ID	Title	Severity	Status
CR-01	Pools Transfer More Tokens than Expected	Critical	Fixed
CR-02	Lost Funds when Burning Yield Tokens	Critical	Fixed
CR-03	Denial of Service when Burning Yield Tokens	Critical	Fixed
ME-01	Denial of Service when Burning Key Tokens	Medium	Fixed
MI-01	Low Decimal Precision	Minor	Acknowledged
MI-02	Funds Taken because of Liquidity Pool Token Repetition	Minor	Acknowledged
MI-03	Swap Limit Setter not Restricted	Minor	Fixed
MI-04	Front-running after Pool Creation	Minor	Fixed
MI-05	New Max Expiry in the Past	Minor	Fixed
EN-01	Unused Math Constants and Functions	Enhancement	Not fixed

Severity Classification

Security risks are classified as follows:

- **Critical:** These are issues that we manage to exploit. They compromise the system seriously. They must be fixed **immediately**.
- **Medium:** These are potentially exploitable issues. Even though we did not manage to exploit them or their impact is not clear, they might represent a security risk in the near future. We suggest fixing them **as soon as possible**.
- **Minor:** These issues represent problems that are relatively small or difficult to take advantage of but can be exploited in combination with other issues. These kinds of issues do not block deployments in production environments. They should be taken into account and be fixed **when possible**.
- **Enhancement:** These kinds of findings do not represent a security risk. They are best practices that we suggest to implement.

This classification is summarized in the following table:

SEVERITY	EXPLOITABLE	ROADBLOCK	TO BE FIXED
Critical	Yes	Yes	Immediately
Medium	In the near future	Yes	As soon as possible
Minor	Unlikely	No	Eventually
Enhancement	No	No	Eventually

Issues Found by Severity

Critical Severity Issues

CR-01 Pools Transfer More Tokens than Expected

The function `add-to-position()` in the contracts `liquidity-bootstrapping-pool.clar` and `fixed-weight-pool.clar` receives as

a parameter how many tokens the user would like to provide to the pool. In order to conform the pool balance, the function calculates the amount of tokens y to be provided based on the amount of token x. As a consequence, the actual value provided to the pool could be greater than the one specified by the user. The result would be the contract moving more tokens than the ones consented by the user.

Recommendation

The actual amount of token y to be provided to the pool should never be greater than the value defined by the user (`dy`). This could be done by adding an `asserts!` that compares those values.

Solution

Fixed according to the recommendation.

CR-02 Lost Funds when Burning Yield Tokens

In a collateral rebalancing pool (`collateral-rebalancing-pool`.clar), liquidity providers get an amount of yield and key tokens equal to the provided liquidity multiplied by `ltv` (Loan-to-Value). When yield tokens are burnt calling `reduce-position-yield()`, the pool is completely rebalanced, swapping all the collateral. Then, for instance, if a user owns 100% of the pool value and burns all his yield tokens, the balance stored in the blockchain will be reduced to zero. However, the user gets transferred only the exact amount of yield token burnt (`balance * ltv`). Since this does not represent the actual value provided, users will realize they lost funds when they try to burn their key tokens and receive zero tokens because the pool is empty.

Recommendation

Subtract from the pool balance the same amount transferred to the user.

Solution

Fixed according to the recommendation.

CR-03 Denial of Service when Burning Yield Tokens

In a collateral rebalancing pool (`collateral-rebalancing-pool`.clar), when yield tokens are burnt calling `reduce-position-yield()`, the pool is completely rebalanced, swapping all the collateral (i.e., it becomes a 0-100 pool). As a consequence, it becomes unusable for token swaps since either the input or output value will be greater than the zero-balance, causing the transaction to revert because of the assertions made in `weighted-equation.clar`. These assertions verify the value deposited to a pool in an A token is lower than that token's balance

multiplied by a ratio, and also the amount of B tokens given in exchange is lower than that token's balance multiplied by a ratio.

Moreover, this Denial of Service would be most of the time unintentionally executed by the users and would persist until someone provides liquidity to the pool again, incrementing the zero-balance.

Solution
Swap functions can only be called before the expiration block.

Medium Severity Issues

ME-01 Denial of Service when Burning Key Tokens

In a collateral rebalancing pool (`collateral-rebalancing-pool.clar`), if all the key tokens were burnt and the reserve pool (`alex-reserve-pool.clar`) does not have tokens in its balance, users might not burn any amount of their yield tokens.

Since `reduce-position-yield()` swaps all the collateral, a new balance of the token (`new-bal-y`) is calculated based on the previous balance and the amount of tokens received after the swap. Then, the function calculates a variable `shares-to-yield` which is equal to the amount of yield tokens the user wants to burn (`shares`) divided by its total supply (`yield-supply`). Afterwards, it calculates `dy`, the product of `new-bal-y` and `shares-to-yield`, and compares `dy` to `shares`. Since the pool only has a value equal to the yield supply, it is swapped and fees are taken out, `dy` will be lower than `shares`. Therefore, in line 496, it calls `alex-reserve-pool.remove-from-balance()` to transfer the shortfall, but if `alex-reserve-pool` does not have tokens, it will revert.

Solution
Changes made to fix CR-02 solved this issue.

Minor Severity Issues

MI-01 Low Decimal Precision

Clarity integer types are stored in 128-bit storage slots. In order to represent the fractional part, a fixed point precision is used with 8 decimal for the fractional part and 30 for the whole part. However, this precision is low and can result in significant decimal errors in the long term.

Recommendation

Since the whole part is more than long enough, the precision could be increased to, for instance, 18 decimal places.

Solution
The development team is studying the implementation of a higher decimal precision.

MI-02 Funds Taken because of Liquidity Pool Token Repetition

Whenever users provide liquidity to a pool, they receive liquidity pool tokens (LP tokens). If two pools were created with the same LP token, users could provide liquidity in one of those pools and withdraw funds from the other. Currently, only the contract owners can create pools and it is at their discretion.

Recommendation
Restrict LP token repetition, not only in each contract, but also across pool contracts: `fixed-weight-pool.clar`, `liquidity-bootstrapping-pool.clar`, `yield-token-pool.clar`. These are the pools that share the same trait for the LP tokens.

Solution
The development team is considering maintaining a registry contract that is updated every time a pool is created and checks if the principal address of the pool token passed is already in use.

MI-03 Swap Limit Setter not Restricted

Amounts swapped are limited in order not to be larger than a certain percentage of total balance (MAX-IN-RATIO and MAX-OUT-RATIO). These values are set initially to 30% in `weighted-equation.clar` and `yield-token-equation.clar`, but they can be modified without restriction. It can result in a denial of service if the value is too small.

Recommendation
Setters should check the new value is, at least, greater than zero with an `asserts!`.

Solution
Fixed according to the recommendation.

MI-04 Front-running after Pool Creation

Liquidity bootstrapping pools (`liquidity-bootstrapping-pool.clar`) are created with a default value for maximum and minimum prices for bootstrapped token (maximum is set to 10^8 and minimum to zero). They can be changed, but only

calling the setter in another transaction. Since the pool creator must provide liquidity in the same transaction they created the pool, under certain market conditions, an attacker might find it profitable to execute a swap immediately after the pool creation and before the maximum and minimum are modified.

Recommendation
Add two parameters to `create-pool()` to assign initial values for `price-x-min` and `price-x-max`.

Solution
Fixed according to the recommendation.

MI-05 New Max Expiry in the Past

In `yield-token-pool.clar`, `max-expiry` is a variable essential to calculate the time to maturity. If this variable is lower or equal to the current block, most of the transactions will revert, causing a denial of service in that contract. Currently, its setter, `set-max-expiry()`, does not check if the new value is valid.

Recommendation
Add an `asserts!` to check `new-max-expiry` is greater than `block-height` before assigning it to `max-expiry`.

Solution
Fixed according to the recommendation.

Enhancements

EN-01 Unused Math Constants and Functions

Useful math functions and constants are copied in the contracts. However, they are not always used by the rest of the contract functions and make the deployment more expensive. For instance, in `collateral-rebalancing-pool.clar`, `pow-up()`, `ONE_10`, `log-fixed()` and `ln-fixed()` are not used.

Recommendation
Remove the functions that are not necessary for the contracts.

Other Considerations

1. `collateral-rebalancing-pool.clar` uses an approximation for the error function (Abramowitz and Stegun) which has a high maximum error. The

recommendation is to extensively test it to assure it does not have unexpected implications.

2. In `yield-token-pool` contract, `max-expiry` value is based on a 15-second block time. AlexGo team explained it is set for the mocknet environment. It should be changed to 10 minutes when it is deployed to Stacks mainnet.

Conclusion

We found the contracts to be simple and straightforward and have an adequate amount of documentation, but not always up-to-date. Three critical, one medium and five minor severity issues were found. Additionally, an enhancement was proposed. The development team fixed seven issues and acknowledged two minor issues.

Disclaimer: This audit report is not a security warranty, investment advice, or an approval of the AlexGo project since CoinFabrik has not reviewed its platform. Moreover, it does not provide a smart contract code faultlessness guarantee.