Allbridge Estrela

Stellar Audit Bank

Reference	24-03-1573-REP
Version	1.1
Date	2024/04/23

Quarkslab

Quarkslab SAS
10 boulevard Haussmann
75009 Paris
France

1. Project Information

Document history			
Version	**Date**	**Details**	**Authors**
1.0	2024/04/03	Initial version	Elouan Wauquier Madigan Lebreton
1.1	2024/04/23	Fixes review	Madigan Lebreton

Quarkslab		
Contact	**Role**	**Contact Address**
Frédéric Raynal	CEO	`fraynal@quarkslab.com`
Pauline Sauder	Project Manager	`psauder@quarkslab.com`
Stavia Salomon	Sales	`ssalomon@quarkslab.com`
Elouan Wauquier	R&D Engineer	`ewauquier@quarkslab.com`
Madigan Lebreton	R&D Engineer	`mlebreton@quarkslab.com`

Allbridge		
Contact	**Role**	**Contact Address**
Yuriy Savchenko	CTO & Co-founder	`ys@allbridge.io`

Contents

1 Project Information **1**

2 Executive Summary **3**
2.1 Context . . . 3
2.2 Objectives . . . 3
2.3 Methodology . . . 3
2.4 Disclaimer . . . 3
2.5 Findings Summary . . . 3
2.6 Recommendations and Action Plan . . . 4
2.7 Conclusion . . . 5
2.8 Fixes . . . 5

3 Manual review **6**
3.1 Factory . . . 6
3.1.1 Purpose . . . 6
3.1.2 Storage . . . 6
3.1.3 Permissioned functionality . . . 7
3.1.4 View functionality . . . 8
3.2 Pool . . . 9
3.2.1 Purpose . . . 9
3.2.2 Storage . . . 9
3.2.3 Permissioned functionality . . . 9
3.2.4 Computations . . . 10
3.2.5 Fees . . . 14

4 Appendix **16**
4.1 Factory contract interface . . . 16
4.2 Pool contract interface . . . 17

2. Executive Summary

2.1 Context

This report presents the work of the collaboration between Allbridge and Quarkslab, as defined in 24-03-1559-PRO . Quarkslab's objective was to conduct a security assessment of two (2) smart contracts for a Soroban decentralized exchange called Estrela.

The audit parameter was defined by the content of the following GitHub repository: allbridge-io/dex-soroban-contracts at commit `56be1f00868f25cd67b07aab132138060406114e`.

2.2 Objectives

The purpose was to discover potential security misconfigurations, weaknesses, and vulnerabilities that can be leveraged or exploited by attackers being able to interact directly with the liquidity pools and the factory. To that end, Quarkslab proposed the following approach:

2.3 Methodology

1. Discovery and set-up phase;
2. Manual code review;
3. Testing;
4. Report, Audit and Project Management.

2.4 Disclaimer

This report reflects the work and results obtained within the duration of the audit for the specified scope in 24-03-1559-PRO as agreed between Allbridge and Quarkslab. Tests are not guaranteed to be exhaustive and the report does not ensure that the application is bug-free.

2.5 Findings Summary

ID	Name	Perimeter
HIGH-1	Overflow risk for some values	Pool contract
LOW-1	Centralization risk for trusted admin	Factory contract
LOW-2	Centralization risk for trusted admin	Pool contract
INFO-1	Pool creation may be capped after several deployments	Factory storage
INFO-2	Overflow in internal function	Common
INFO-3	Full fees on liquidity withdrawal	Pool contract fee system

INFO-4	Code duplication for sending rewards	Pool contract reward computation

Severity: critical, high, medium, low, info

2.6 Recommendations and Action Plan

Recommendations

ID	Recommendations	Perimeter
HIGH-1	Explicitely use overflow-safe operations on U256, such as `U256::checked_add` and `U256::checked_mul`. Also limit A to be in a safe range on initialization and configuration.	Pool contract
LOW-1	Consider limiting the amount of trust allocated to the `Admin`. The issue can be mitigated through several ways such as delegating the `Admin` role to a Decentralized Autonomous Organization (DAO) mechanism or to a multi-signature wallet. *Note: The provided mitigations decrease the likelihood of the issue. Decreasing the impact doesn't seem to be possible without removing the functionality.*	Factory contract
LOW-2	Consider limiting the amount of trust allocated to the `Admin`. The issue can be mitigated through several ways such as delegating the `Admin` role to a Decentralized Autonomous Organization (DAO) or to a multi-signature wallet. *Note: The provided mitigations decrease the likelihood of the issue. Decreasing the impact doesn't seem to be possible without removing the functionality.*	Pool contract
INFO-1	Consider limiting the number of deployed pools. *Note: Considering the low number of known stablecoins, this issue is unlikely to occur.*	Factory storage
INFO-2	Use `U256` when multiplying large integers.	Common
INFO-3	Do not apply a fee when withdrawing liquidity.	Pool contract fee system
INFO-4	Remove code duplication.	Pool contract reward computation

Severity: critical, high, medium, low, info

2.7 Conclusion

With a trusted administrator (e.g. a DAO) and common configuration parameters, the pool is safe from exploits to the best of our knowledge.

However, when configured with a high amplification parameter ($A \geq 121$), the DEX may be vulnerable to a theft of its liquidity.

In addition, we expressed recommendations on the fee system and some coding practices.

2.8 Fixes

On the 2024/04/22, Quarkslab reviewed the fixes implemented following the reported vulnerabilities. The reviewed commit is `ddf0b18289dbcc7631b97551531eb8ef79d5ffbf`.

ID	Name	Fix status
HIGH-1	Overflow risk for some values	✓
LOW-1	Centralization risk for trusted admin	✗
LOW-2	Centralization risk for trusted admin	✗
INFO-1	Pool creation may be capped after several deployments	✓
INFO-2	Overflow in internal function	✓
INFO-3	Full fees on liquidity withdrawal	✗
INFO-4	Code duplication for sending rewards	✓

Severity: critical, high, medium, low, info
Fix status: ✗ acknowledged, ∼ mitigated, ✓ fixed

LOW-1 and LOW-2 were acknowledged by Allbridge, the administrator of the contract needs to be trusted.

INFO-3 was acknowledged by Allbridge, fees on liquidity withdrawal are expected.

3. Manual review

3.1 Factory

3.1.1 Purpose

The factory contract allows its administrator to deploy and initialize liquidity pool for a pair of tokens.

3.1.2 Storage

The factory smart contract stores the association of the pair of tokens (often called ”token A” and ”token B”) and its corresponding deployed liquidity pool.

At the *Instance* level, it remembers the following data:

- the address of the administrator, with the symbol `"Admin"`;
- the factory's specific data, with the symbol `"FactoryInfo"`;

The `FactoryInfo` structure stores the factory's specific data. Two fields are defined in this specific structure:

- `wasm_hash`: a 32-byte hash corresponding to the pool's code hash;
- `pairs`: a map that associates a pair of tokens to its deployed pool contract;

The Instance level is appropriate for the configured fields.

`INFO`	`INFO-1` Pool creation may be capped after several deployments
Perimeter	Factory storage
Fix status	✓

Description

The Soroban documentation states that a *”ledger entry is read or written from the ledger in its entirety; there is no way to read or write ”only a part” of a CONTRACT_DATA ledger entry”.*
In the case of the factory, it means that the `pairs` map stored in the `FactoryInfo` ledger entry may not be readable or writable if too many pools are deployed.

Recommendation

Consider limiting the number of deployed pools.
Note: Considering the low number of known stablecoins, this issue is unlikely to occur.

The issue was fixed by capping the number of pairs to 20.

3.1.3 Permissioned functionality

- The factory's `initialize` method can be called only when the `FactoryInfo` structure in Instance storage does not exist. Since the TTL of the contract instance and all globals are tied together, there is no risk of this field expiring before the contract instance.

  Anyone can call this function, but it can be called only once overall. It configures the contract's administrator and initializes the `FactoryInfo` with the `wasm_hash` parameter.

  

  The `initialize` method should be called first, and the contract should not be used unless a trusted party has successfully called this function.

  Once the contract is initialized, all the modifying state functions are access controlled. They check that the configured administrator has authorized the transaction before modifying the contract's state.

- `create_pair` lets the administrator deploy a new pool contract for a pair of tokens.
- `set_admin` allows the administrator to transfer ownership of the contract to another address.
- `update_wasm_hash` lets the administrator modify the `wasm_hash` used during pool deployment. It will update the WASM code of the pool contract. Modifying this parameter will not impact the pools already deployed, only the new pools will be affected.
- `upgrade` is available to update the factory executable WASM. The administrator can upgrade the logic of this contract. The new executable will be updated after the current invocation, in case it runs successfully

LOW	LOW-1 Centralization risk for trusted admin	
Likelihood	●○○○	**Impact** ●●○○
Perimeter	Factory contract	
Prerequisites	`Admin` role	
Fix status	✗	

Description

Factory has an `Admin` address with privileged rights to perform admin tasks such as pool contract's code update. `Admin` needs to be trusted to not perform malicious updates.
A malicious update of the pool contract's hash leads to the deployment of malicious pool contracts. Users' funds may be at risk in newly deployed pools.

Recommendation

Consider limiting the amount of trust allocated to the `Admin`. The issue can be mitigated through several ways such as delegating the `Admin` role to a Decentralized Autonomous Organization (DAO) mechanism or to a multi-signature wallet.
Note: The provided mitigations decrease the likelihood of the issue. Decreasing the impact doesn't seem to be possible without removing the functionality.

The issue was acknowledged.

3.1.4 View functionality

Four (4) view functions are defined in the factory smart contract. These functions are permissionless, they let users retrieve information about the state of the protocol.

- `pool` takes a pair of tokens as input and returns the associated deployed pool contract. If none exists, an `Error::NotFound` is returned.
- `pools` returns a map of all the deployed pools with their associated pair of tokens.
- `get_wasm_hash` returns the configured WebAssembly code hash if it exists, an error otherwise.
- `get_admin` returns the current administrator address if it exists, an error otherwise.

3.2 Pool

3.2.1 Purpose

The pool is an Automatic Market Maker (AMM) using the StableSwap invariant. It holds liquidity and provides a market price for two tokens (supposedly stablecoins).

Liquidity providers mint virtual LP tokens when providing liquidity (representing their share of the pool's balance), but these tokens are not real tokens (e.g. not transferable). These are burnt when the corresponding liquidity is withdrawn.

3.2.2 Storage

The pool stores the details of liquidity providers and the state of the pool. We found the choice of storage level appropriate for the types of data stored.

Instance

At the *Instance* level, the pool remembers the following data:

- the address of the administrator, with the symbol "Admin";
- the pool's specific data, with the symbol "Pool", i.e.,
  - token related data in `tokens`, `tokens_decimals`, and `token_balances`;
  - the amount of liquidity as `total_lp_amount`;
  - the fee schedule as `fee_share_bp` and `admin_fee_share_bp` (by increments of 0.01% since `BP` $= 10\,000$);
  - fee reward amounts in `acc_rewards_per_share_p` and `admin_fee_amount`; and
  - the StableSwap "amplify" parameter (A) as `a`.

All these fields have a fixed size, thus cannot grow uncontrollably.

Persistent

At the *Persistent* level, the pool stores `UserDeposit`s, containing the amount of liquidity deposited (as `lp_amount`) and the amount of rewards claimed (as `reward_debts`).

Temporary

No *Temporary* storage is used.

3.2.3 Permissioned functionality

- The pool's `initialize` method can be called only when the `Pool` structure in Instance storage does not exist. Since the TTL of the contract instance and all globals are tied together, there is no risk of this field expiring before the contract instance.

Anyone can call this function, but it can be called only once overall. It configures the administrator of the contract and initializes the `Pool`.

The `initialize` method should be called first, and the contract should not be used unless a trusted party has called this function successfully. When the factory creates a new pool, this function is automatically called.

- `set_admin` allows the administrator to transfer ownership of the contract to another address.
- `set_admin_fee_share` and `set_fee_share` allow the administrator to configure the pool's fee rates.
- `upgrade` is available to update the pool's code. The administrator can upgrade the logic of this contract. The new executable will be updated after the current invocation, in case it runs successfully.

LOW	LOW-2 Centralization risk for trusted admin		
Likelihood	●○○○	**Impact**	●●○○
Perimeter	Pool contract		
Prerequisites	`Admin` role		
Fix status	✗		

Description

Factory has an `Admin` address with privileged rights to perform admin tasks such as pool contract's code update. `Admin` needs to be trusted to not perform malicious updates.
A malicious update of the pool contract's code can allow an admin to pull user's funds.

Recommendation

Consider limiting the amount of trust allocated to the `Admin`. The issue can be mitigated through several ways such as delegating the `Admin` role to a Decentralized Autonomous Organization (DAO) or to a multi-signature wallet.
Note: The provided mitigations decrease the likelihood of the issue. Decreasing the impact doesn't seem to be possible without removing the functionality.

The issue was acknowledged.

3.2.4 Computations

The pool uses the StableSwap invariant developed for the Curve protocol. It ensures the pool always has some liquidity for swapping assets while maintaining a low slippage, and is designed for stablecoins.

The invariant is

$$An^n \sum_{i=1}^{n} x_i + D = ADn^n + \frac{D^{n+1}}{n^n \prod_{i=1}^{n} x_i},$$

where

- D is "the total amount of coins when they have an equal price".
- A is an "amplification coefficient" akin to a leverage when the pool is balanced. When the pool is imbalanced, it loses its power and leverage decreases. The actual "leverage" value is $\chi = A\frac{\prod_{i=1}^{n} x_i}{(D/n)^n}$, with $\prod_{i=1}^{n} x_i = \left(\frac{D}{n}\right)^n$ when the pool is balanced. A should be tuned according to the expected liquidity and volatility of the tokens.

Allbridge uses pools with only 2 assets (denoted x and y), turning the invariant into

$$4A(x+y) + D = 4AD + \frac{D^3}{4xy}$$

All functions need to maintain this invariant at least (except for rounding and fees).

We evaluated the mathematical functions for correctness and paid attention to the following details.

- The smart contract is compiled with `profile.release.overflow-checks = true`, thus arithmetic operations may crash the smart contract.
- Even with the above flag set, `ethnum::U256` *overflows silently* (i.e. wraps), even in release mode.
- The logical shift operators `>>` and `<<` are exempt from overflow checks. This has no impact for right shifts but may cause unwanted overflows for left shifts.
- Values are not real integers, rather they are fixed point numbers which decimal count can be token-specific or the `SYSTEM_PRECISION = 3`. Multiplications, divisions and roots can affect the dimension of these numbers.

Utilitary functions

When solving the invariant equation, the pool needs to compute square and cubic roots. These are defined in `common/shared/src/utils/num.rs`.

We verified that `pub fn sqrt(n: &U256) -> U256` computes $\lfloor\sqrt{n}\rfloor$ using fuzzing and manual code review. The left shift is bounded to `1 << 254`. No arithmetic operations risk overflowing.

We verified that `pub fn cbrt(n: &U256) -> u128` computes $\lfloor\sqrt[3]{n}\rfloor$ using fuzzing and manual code review. Since $\sqrt[3]{\texttt{U256::MAX}} < \sqrt[3]{2^{256}} = 2^{256/3} < 2^{86} < \texttt{u128::MAX}$, the return type can fit the result value. For the same reason, the left shift cannot overflow. However, on line 29, the computation of `hi * hi` may overflow (i.e. `panic!()`) for large values of `n` ($n \geq 2^{192}$).

We verified this by adding this test to `common/shared/src/utils/num.rs`:

```
#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    #[should_panic(expected = "attempt to multiply with overflow")]
    fn cbrt_panic() {
        cbrt(&(U256::ONE << 192));
    }
}
```

We traced the (only) calling site, and found no way to reach this issue because of the `pub(crate) const MAX_TOKEN_BALANCE: u128 = 2u128.pow(40)` constraint, hence the lower impact score.

INFO	INFO-2 Overflow in internal function
Perimeter	Common
Fix status	✓
Description	
The `cbrt` function can panic for some values within its domain. This cannot be reached.	
Recommendation	
Use `U256` when multiplying large integers.	

The issue was fixed by handling the error.

Invariant computation

During the pool's operation, x, y and D vary. To maintain the invariant, each of these variables need to be expressed in terms of the others.

- A is set at initialization time;
- D is the amount of liquidity (in `total_lp_amount`);
- x and y are the token balances with system precision (in `token_balances` and `token_decimals`).

The amount of liquidity D is computed in `pub fn get_d(&self, x: u128, y: u128) -> u128`.

Given its upper bound of 2^{40}, we found no way to lock the smart contract or perform incorrect computations for reasonable values of A. However, we still recommend checking for overflows manually in `ethnum::U256`.

The x and y variable play equivalent roles, so a single function is required:
`pub fn get_y(&self, native_x: u128, d: u128) -> Result<u128, Error>`.

For $A \geq 121$, we found it possible to trigger an overflow on line 280.

HIGH	HIGH-1 Overflow risk for some values	
Likelihood	●●○○	**Impact** ●●●●
Perimeter	Pool contract	
Prerequisites	$A \geq 121$	
Fix status	✓	

Description

An overflow can happen in `Pool::get_d` when depositing liquidity. This can lead to incorrect bookkeeping of the pool's liquidity tokens, increasing their value. In turn, this enables an attacker to withdraw more tokens than they deposited.

Recommendation

Explicitely use overflow-safe operations on U256, such as `U256::checked_add` and `U256::checked_mul`.
Also limit A to be in a safe range on initialization and configuration.

The issue was fixed by capping the value of A to 60 and by using overflow-safe operations.

Swaps

The `swap` function uses `get_receive_amount` to compute the new token balances and rewards.

The user can specify the minimum amount they expect to receive. This mitigates the effects of volatility and sandwich attacks, which is common practice for automatic market makers.

Deposit

The `deposit` function uses `get_deposit_amount` to compute the new balances and the updated amount of LP tokens. It then collects the stablecoins and delivers the LP tokens to the user. Finally, it hands the user their pending rewards (as if they had called `claim_rewards`, see the section on rewards).

Withdraw

The `withdraw` function uses `get_withdraw_amount` to compute the withdrawn amounts and fees. It then transfers the tokens to the user along with pending rewards (as if they had called `claim_rewards`, see the section on rewards), and burns the specified amount of LP tokens.

INFO	INFO-3 Full fees on liquidity withdrawal
Perimeter	Pool contract fee system
Fix status	✗
Description	
Liquidity providers pay fees on the full liquidity they withdraw, which may be unexpected by users.	
Recommendation	
Do not apply a fee when withdrawing liquidity.	

The issue was acknowledged, it is expected behavior.

Rewards

The `claim_rewards` function computes and transfers the pending rewards for a user. Pending rewards are also computed and sent separately when withdrawing and depositing liquidity.

INFO	INFO-4 Code duplication for sending rewards
Perimeter	Pool contract reward computation
Fix status	✓
Description	
Pending rewards for liquidity providers are computed separately in `get_pending` and `claim_rewards`. The total reward share of liquidity providers are computed separately in `get_pending` and `get_reward_debts`.	
Recommendation	
Remove code duplication.	

The issue was fixed by removing duplicated code.

3.2.5 Fees

The pool collects two types of fees:

- regular fees (configured in `Pool::fee_share_bp`): a percentage of every swap and *every withdrawal*.
- admin fees (configured in `Pool::admin_fee_share_bp`): a percentage of the regular fees.

During initialization and pool configuration, the fees can be set to $[0, 100\%)$:

$$\frac{k}{\texttt{Pool} :: \texttt{BP}}, \text{ where } k \in [\![0, \texttt{Pool} :: \texttt{BP} - 1]\!] \text{ and } \texttt{Pool} :: \texttt{BP} = 10\,000.$$

Admin fees accumulate in `Pool::admin_fee_amount`, which is reset to 0 when the admin collects the fees.

Liquidity provider fees accumulate in `Pool::acc_rewards_per_share_p`. When a liquidity provider collects their fee, the total amount they collected so far is saved in their `UserDeposit::reward_debts` and cannot be collected again.

Rewards are tracked seperately for both tokens, so that variations in the pool's balance does not impact past rewards.

4. Appendix

4.1 Factory contract interface

External function	Access control	Operations
`initialize`	✗	Read, Write
`create_pair`	✓	Read, Write
`set_admin`	✓	Read, Write
`update_wasm_hash`	✓	Read, Write
`upgrade`	✓	Read, Write
`pool`	✗	Read-only
`pools`	✗	Read-only
`get_wasm_hash`	✗	Read-only
`get_admin`	✗	Read-only

4.2 Pool contract interface

External function	Access control	Operations
`initialize`	✗	Read, Write
`upgrade`	✓	Read, Write
`set_admin`	✓	Read, Write
`set_admin_fee_share`	✓	Read, Write
`set_fee_share`	✓	Read, Write
`deposit`	✗	Read, Write
`withdraw`	✗	Read, Write
`swap`	✗	Read, Write
`claim_rewards`	✗	Read, Write
`pending_reward`	✗	Read-only
`get_pool`	✗	Read-only
`get_user_deposit`	✗	Read-only
`get_d`	✗	Read-only
`get_receive_amount`	✗	Read-only
`get_send_amount`	✗	Read-only
`get_withdraw_amount`	✗	Read-only
`get_deposit_amount`	✗	Read-only
`get_admin`	✗	Read-only