Allbridge Estrela

Stellar Audit Bank

Reference 25-01-1969-REP
Version 1.1
Date 2025-02-05

Quarkslab SAS
10 boulevard Haussmann
75009 Paris
France

1. Project Information

1.1. Document history

Version	Date	Details	Authors
1.0	2025-01-17	Initial version	Elouan Wauquier
1.1	2025-02-05	Corrected version following *Allbridge*'s feedback	Elouan Wauquier

1.2. Contacts

1.2.1. Quarkslab

Contact	Role	Email
Frédéric Raynal	CEO	fraynal@quarkslab.com
Pauline Sauder	Project Manager	psauder@quarkslab.com
Stavia Salomon	Sales	ssalomon@quarkslab.com
Elouan Wauquier	R&D Security Engineer	ewauquier@quarkslab.com

1.2.2. Client

Contact	Role	Email
Yuriy Savchenko	CTO & Co-Founder	ys@allbridge.io
Pavel Velykyi	CBDO	pv@allbridge.io

Contents

2. Executive Summary

2.1. Context

Allbridge engaged with Quarkslab to perform a security audit of the smart contracts related to Estrela, as defined in 25-01-2923-PRO. This product is a decentralized exchange for stablecoins implemented as an Automated Market Maker (AMM).

Users of *Allbridge* Estrela can provide liquidity to pools of 2 or 3 tokens, and swap one of these tokens for any other.

Quarkslab had already performed an audit of an earlier version of these smart contracts, with support for only 2 tokens per pool.

The audit parameter was defined by the content of the following GitHub repository: allbridge-io/dex-soroban-contracts at commit `dd8d678b9012dc55624f16e1693ee77b048686cd`.

2.2. Objectives

The objective of the audit is to ensure that, among other things:

- the pool invariant is maintained — notably, solutions to the invariant equation are correct and well implemented;
- one cannot steal tokens from liquidity providers;
- liquidity providers cannot lose their liquidity, except for impermanent loss;
- one cannot manipulate rewards;
- one cannot prevent someone else from earning rewards or using the platform;
- one cannot generally abuse, take control, or drain smart contracts from *Allbridge* Estrela.

2.3. Methodology

Allbridge provided Quarkslab with the source code and tests for their smart contracts. To assess their security, we first needed to familiarize ourselves with the project's structure, and to understand the key tasks outlined in the audit scope. We carefully reviewed the project resources to gain a clear understanding of the features to be evaluated. Armed with this understanding, we started reviewing the smart contracts, and conducted tests to confirm our findings.

Overall, the following steps were defined for the security audit:

1. **Manual code review**
   - Identify the weaknesses and misbehaviors in the code, that could lead to vulnerabilities;
   - Identify and exploit the vulnerabilities specific to smart contracts (reentrancy, frontrunning, external contract calls, gas constraints…);
   - Focus on critical functions and key aspects, including permissions, access control, logical bugs, and erroneous computations;

- Check the implementation for best practices (e.g., critical condition validation, documentation/specification, storage structure, trust assumptions…).

2. **Unit and functional tests**
   - Ensure that the smart contract behaves as we understood it during step 1;
   - Test the interactions with other smart contracts and users in realistic scenarios.
3. **Report edition**
   - Write a report including the assessment methodology, findings and recommendations.

2.4. Findings Summary

During the time frame of the security audit, Quarkslab discovered only 1 issue considered as low severity, and 3 issues considered informative.

Info

This report reflects the work and results obtained within the duration of the audit and on the specified scope, as agreed between *Allbridge* and Quarkslab. Tests are not guaranteed to be exhaustive, and the report does not ensure that the code is bug- or vulnerability-free.

ID	Name	Perimeter
LOW-2	First deposit can be backrun.	Pool
INFO-1	Centralization risk for the administrator.	
INFO-3	Token called twice on deposit.	Pool
INFO-4	Generic `calc_withdraw_amount()` assumes a specific missing index.	Pool

2.5. Conclusions

To the best of our knowledge, the smart contract logic is sound, the implementation of the StableSwap invariant is correct and cannot be exploited with the current parameter constraints. The only security issue we found has low impact and only affects the first deposit.

3. Reading Guide

This reading guide describes the different sections present in this report and gives some insights about the information contained in each of them and how to interpret it.

3.1. Executive summary

The executive summary Section 2 presents the results of the assessment in a non-technical way, summarizing all the findings and explaining the associated risks. For each vulnerability, a severity level is provided as well as a name or short description, and one or more mitigation, as shown below.

ID	Name	Category
CRITICAL	Vulnerability Name #1	Injection
HIGH	Vulnerability Name #2	Remote code injection
MEDIUM	Vulnerability Name #3	Denial of Service
LOW	Vulnerability Name #4	Information leak

Each vulnerability is identified throughout this document by a unique identifier `<LEVEL>-<ID>`, where `ID` is a number and `LEVEL` the severity (`INFO`, `LOW`, `MEDIUM`, `HIGH` or `CRITICAL`). Every vulnerability identifier present in the vulnerabilities summary table is a clickable link that leads to the corresponding technical analysis that details how it was found (and exploited if it was the case). Severity levels are explained in Section 3.2.

The executive summary also provides an action plan with a focus on the identified *quick wins*, some specific mitigation that would drastically improve the security of the assessed system.

3.2. Metric definition

This report uses specific metrics to rate the severity, impact and likelihood of each identified vulnerability.

3.2.1. Impact

The impact is assessed regarding the information an attacker can access by exploiting a vulnerability but also the operational impact such an attack can have. The following table summarizes the different levels of impact we are using in this report and their meanings in terms of information access and availability.

CRITICAL	Allows a total compromise of the assessed system, allowing an attacker to read or modify the data stored in the system as well as altering its behavior.
HIGH	Allows an attacker to impact significantly one or more components, giving access to sensitive data or offering the attacker a possibility to pivot and attack other connected assets.
MEDIUM	Allows an attacker to access some information, or to alter the behavior of the assessed system with restricted permissions.
LOW	Allows an attacker to access non-sensitive information, or to alter the behavior of the assessed system and impact a limited number of users.

3.2.2. Likelihood

The vulnerability likelihood is evaluated by taking the following criteria in consideration:

- **Access conditions**: the vulnerability may require the attacker to have priviledged access, or to have additional information.
- **Required skills**: an attacker may need specific skills to exploit the vulnerability.
- **Known available exploit**: when a vulnerability has been published and an exploit is available, the probability a non-skilled attacker would find it and use it is pretty high.

The following table summarizes the different level of vulnerability likelihood:

CRITICAL	The vulnerability is easy to exploit even from an unskilled attacker and has no specific access conditions.
HIGH	The vulnerability is easy to exploit but requires some specific conditions to be met (specific skills or access).
MEDIUM	The vulnerability is not trivial to discover and exploit, requires very specific knowledge or specific access (internal network, physical access to an asset).
LOW	The vulnerability is very difficult to discover and exploit, requires highly specific knowledge or authorized access

3.2.3. Severity

The severity of a vulnerability is defined by its impact and its likelihood, following the following table:

		Impact			
		●●●●	●●●○	●●○○	●○○○
Likelihood	●●●●	CRITICAL	CRITICAL	HIGH	MEDIUM
	●●●○	CRITICAL	HIGH	HIGH	MEDIUM
	●●○○	HIGH	HIGH	MEDIUM	LOW
	●○○○	MEDIUM	MEDIUM	LOW	LOW

4. Manual review of smart contracts

The codebase's central smart contract is that of a decentralized exchange pool for stablecoins, designed as an Automated Market Maker (AMM) using CurveFinance's StableSwap invariant. It is defined generically over the number of supported tokens, and 2 specialized implementations are provided: one for 2-tokens pools, and another for 3-tokens pools.

A factory is also defined to deploy the pool instances.

Description	Crate path
Utility and shared definitions, complementing the Soroban SDK.	`common/shared`
Procedural macros for the above definitions.	`common/proc_macros`
Generic definition of an Automated Market Maker (AMM), with a procedural macro for specialization.	`common/generic_pool`
Storage and authentication of the administrator's address.	`common/storage`
Factory smart contract to deploy pool instances.	`contracts/factory`
Specialization of the generic pool for 2 tokens.	`contracts/two_pool`
Specialization of the generic pool for 3 tokens.	`contracts/three_pool`

Table 1: The codebase's file structure.

4.1. Factory

The factory contract lets its administrator deploy and initialize pools for a pair or triplet of tokens.

4.1.1. Storage

Storage structures are defined with custom procedural macros, handling storage location, key, and time to live (TTL) extension. TTL extension is performed automatically on access.

Key	Location	Structure
Constant	Instance	`struct Admin(Address)`
Constant	Instance	`struct FactoryInfo { two_pool_wasm_hash: BytesN<32>, three_pool_wasm_hash: BytesN<32>, pools: Map<Vec<Address>, Address>, }`

Note that all storage variables are located in *Instance*, including the list of pools. However, since this list is capped (to `MAX_PAIRS_NUM = 21`), the factory Instance storage cannot become too expensive.

4.1.2. Functionality

The factory's Application Binary Interface (ABI) is given in Appendix A.

4.1.2.1. Initialization

The factory's `initialize()` function can only be called 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 this field expires before the contract instance.

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

4.1.2.2. Configuration

At any time, the administrator can modify the implementation of the factory itself (using `upgrade()`), or that of future deployed pools (using `update_two_pool_wasm_hash()` or `update_three_pool_wasm_hash()`). They can also transfer their administrative rights to another address with `set_admin()`.

We did not find any issue with the implementation of these functions.

4.1.2.3. Pool creation

The administrator can call `create_pool()` with an `Vec` of 2 or 3 tokens, and the pool's initial configuration.

The factory checks all input parameters, sorts the tokens to avoid creating a duplicate pool, and deploys the pool at with a salt depending on itself and the token `Vec`.

It then initializes the pool with the provided parameters.

4.2. Pool

The pool is an Automated Market Maker (AMM) using CurveFinance's StableSwap invariant. It holds liquidity and provides a market price for `N` stablecoins.

Liquidity Providers (LPs) mint virtual LP tokens when providing liquidity, representing their share of the pool's balance. These tokens are not fully fledged tokens (e.g. non-transferable). They are burnt when the corresponding liquidity is withdrawn.

Users can swap one stablecoin for another at any time, following the invariant. A fee is paid to LPs when swapping or withdrawing liquidity. An admin fee is also taken on that fee.

4.2.1. Storage

Storage structures are defined with custom procedural macros, handling storage location, key, and TTL extension. TTL extension is performed automatically on access.

The `SizedAddressArray`, `SizedU128Array`, and `SizedDecimalsArray` types are simply wrappers around `Vec<Address>`, `Vec<u128>`, and `Vec<u32>` (respectively) with a `usize` index instead of `u32`, and an `impl Deref`.

Key	Location	Structure	Comments
Constant	Instance	`struct Admin(Address)`	
Constant	Instance	`struct PoolInfo { a: u128, fee_share_bp: u128, admin_fee_share_bp: u128, total_lp_amount: u128, tokens: SizedAddressArray, tokens_decimal: SizedDecimalArray, token_balances: SizedU128Array, acc_rewards_per_share_p: SizedU128Array, admin_fee_amount: SizedU128Array, }`	• `a` is the amplification factor A in the invariant equation; • `total_lp_amount` corresponds to D in the invariant equation; • fees have a precision of 0.01%; • the 5 arrays have a length of `N` and index tokens in a Structure of Array (SoA) layout; • the accumulated rewards should be excluded from `token_balances`.
`Address`	Persistent	`struct UserDeposit { lp_amount: u128, reward_debts: SizedU128Array, }`	• `reward_debts` keeps track of the claimed rewards.

4.2.2. Functionality

The pool's Application Binary Interface (ABI) is given for reference in Appendix B.

4.2.2.1. Initialization

The pool's `initialize()` function can only be called when the `PoolInfo` structure in *Instance* storage does not exist. Since the TTL of the contract instance and all globals are tied together, there is no risk this field expires before the contract instance.

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

The fees are capped to a maximum of 100% (more would be nonsensical). A is capped to `60` to avoid overflows in computations. The tokens's decimals are retrieved by calling the tokens directly at initialization time. Note that no liquidity is present in the pool just after initialization.

Info

The `initialize()` function 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.

4.2.2.2. Configuration

At any time, the administrator can modify the pool's fees (up to 100%) using `set_fee_share()` or `set_admin_fee_share()`; or transfer the administrative rights to another address with `set_admin()`. Since they can also upgrade the smart contract, they can also alter any aspect of it.

INFO-1	Centralization risk for the administrator.
Prerequisite	`Admin` role

The pool and factory smart contracts have an `Admin` address with privileged rights to perform administrative tasks, such as code upgrade. The administrator needs to be trusted not to perform malicious upgrades. A malicious upgrade of the pool contract's code can allow an administrator to steal the LPs's funds.

Governance mechanisms are outside the scope of this audit, so we could not assign a severity score. *Allbridge* already acknowledged the issue during our last audit with them.

Recommendation

Consider limiting the amount of trust allocated to the administrator. The finding can be mitigated in several ways, such as delegating the `Admin` role to a Decentralized Autonomous Organisation (DAO), or to a multi-signature wallet.

The administrator can also claim their administrator rewards with `claim_rewards()`.

We did not find any issue with the implementation of these functions.

4.2.2.3. Deposit

When providing liquidity, the user sends tokens to the pool and mints LP tokens in exchange. The amount of LP tokens minted is the difference between the theoretical value of D after the deposit, and its stored value before the deposit (in `total_lp_amount`). After the deposit, `total_lp_amount` is thus equal to its theoretical value.

The first deposit (when there is no liquidity) needs to be an equal amount of each tokens.

LOW-2	First deposit can be backrun.		
Likelihood	●○○○	**Impact**	●○○○
Prerequisite	Depegged token		

After the first deposit, the pool has the same amount of each token, thus it considers all tokens to have the exact same value. In case one (or more) stablecoin(s) is not at its peg, the gap with its peg can be arbitraged by backrunning the first deposit, causing impermanent loss to the LP.

Under normal circumstances, a large price difference is unlikely for stablecoins since they are supposed to represent the same value.

Recommendation

Allow the initial deposit to be imbalanced, within a reasonable tolerance.

When updating the user's deposit, their rewards are automatically claimed, and the `reward_debts` field is updated with the additional LP tokens. This prevents the newly acquired LP tokens to earn past rewards.

INFO-3	Token called twice on deposit.

When depositing tokens, the smart contract first calls each token to receive the provided amount. Then, it calls each token a second time to send the accumulated rewards. On top of consuming more gas, this can prevent a user from providing the maximum liquidity they can in a single transaction.

Note that both calls are combined when withdrawing tokens.

Recommendation

Combine both calls into one.

4.2.2.4. Withdraw

When withdrawing liquidity, the user burns their LP tokens and receives the corresponding share of each token. A fee is applied on all tokens.

When updating the user's deposit, their rewards are automatically claimed, and the `reward_debts` field is updated with the new balance.

The exact amount to withdraw is computed in the `calc_withdraw_amount()` function. This function considers that the withdrawn amount depends linearly on the share of LP tokens, except for the second least valuable token, which is computed from the invariant.

INFO-4	Generic `calc_withdraw_amount()` assumes a specific missing index.

The generic implementation of `calc_withdraw_amount()` takes a (specific) parameter called `y_indexes`, which dictates which tokens are considered in the invariant computation. For example, it is `&[0]` in the case of `two_pool`, and `&[0, 2]` in the case of `three_pool` (notice the missing `1` in both cases). However, the function assumes that `1` is always the missing index:

common/generic_pool/src/utils.rs — Rust

```
90  token_amounts_sp[1] = pool.token_balances().get(indexes[1]) - y;
```

Recommendation

Only give the index to solve the invariant for as a parameter, and dynamically compute the other ones from `N`.

4.2.2.5. Swap

When swapping one token for another, the pool computes the amount to receive by solving the invariant equation. It is possible to send the output token to any address.

Fees are taken from the output token.

4.2.2.6. Rewards

The pool collects two types of fees:

- regular fees (configured in `Pool::fee_share_bp`): a percentage of the output token in swaps, and every token in withdrawals.
- administrator fees (configured in `Pool::admin_fee_share_bp`): a percentage of the regular fees.

The fees can be set by the administrator to any value in $[0\%, 100\%)$ with increments of 0.01%.

Administrator fees accumulate in `Pool::admin_fee_amount`, which is reset to 0 when the administrator collects the fee by calling `claim_admin_fee()`.

Liquidity provider fees accumulate in `PoolInfo::acc_rewards_per_share_p`. When a liquidity provider claims their fee with `claim_rewards()` (or with `deposit()` or `withdraw()`), the total

amount they are entitled to is saved in their `UserDeposit::reward_debts`, and cannot be claimed again.

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

4.2.2.7. Solving the invariant

CurveFinance's StableSwap invariant is defined as

$$An^n \sum x_i + D = ADn^n + \frac{D^{n+1}}{n^n \prod x_i},$$

where

- x_i are the token balances, for $i \in [\![1, n]\!]$;
- 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 $\chi = \frac{A \prod x_i}{(D/n)^n}$, with $\prod x_i = \left(\frac{D}{n}\right)^n$ when the pool is balanced. A should be tuned according to the expected liquidity and volatility of the tokens.

All functions need to maintain this invariant *at least* (i.e., rounding and fees should only increase its value).

Note

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 with a decimal count of `SYSTEM_PRECISION = 3`. Multiplications, divisions and roots can affect the dimension of these numbers.

Allbridge implements the invariant solutions for $n = 2$ and $n = 3$ tokens.

Two functions provide solutions to the invariant equation:

- `get_d()` solves for D given x_i, and
- `get_y()` solves for one x_i given D and the other x_i.

The 2-tokens pool provides an analytical solution to both functions, using Cardano's formula for `get_d()` and the quadratic formula for `get_y()`.

The 3-tokens pool provides an analytical solution to `get_y()`. For `get_d()`, the invariant equation is a 4-th degree polynomial.

The pool performs a search using Newton's method, with the stopping condition $f < df \iff \frac{f}{df} < 1$, corresponding to the point where no further improvement can be made given the precision.

The initial guess is $\sum x_i$, which is a good guess in this context since it is the value of D when the pool is balanced. The function can be expressed in the form $f(D) = -D^4 - pD + q$, with $p > 0$, and $q > 0$. It thus has two real roots, one positive and one negative. Because it is continuous, has no inflection point, and its extremum is negative, it is guaranteed to converge to the positive root monotonically. With $q \neq 0$, the positive root has multiplicity one, so the convergence rate is quadratic. $A \neq 0 \implies df \neq 0$, so $\frac{f}{df}$ cannot panic.

We did not find a way to exploit the implementation of these functions, given the constraints on A and D (`MAX_A` and `MAX_TOKEN_BALANCE`).

A. Factory ABI

The Application Binary Interface (ABI) is defined in `contracts/factory/src/contract.rs`. Every public function extends the instance's time to live (TTL), except those that do not modify the state (the so-called "view" functions), `initialize()` and `upgrade()`.

Signature	Access control	Immutable

`initialize(`
`  two_pool_wasm_hash: BytesN<32>,`
`  three_pool_wasm_hash: BytesN<32>,`
`  admin: Address,`
`)`
	Only once	✗

`create_pool(`
`  deployer: Address,`
`  pool_admin: Address,`
`  a: u128,`
`  tokens: Vec<Address>,`
`  fee_share_bp: u128,`
`  admin_fee_share_bp: u128,`
`)`
	`admin`	✗

`set_admin(`
`  new_admin: Address,`
`)`
	`admin`	✗

`update_two_pool_wasm_hash(`
`  new_wasm_hash: BytesN<32>,`
`)`
	`admin`	✗

`update_three_pool_wasm_hash(`
`  new_wasm_hash: BytesN<32>,`
`)`
	`admin`	✗

`upgrade(`
`  new_wasm_hash: BytesN<32>,`
`)`
	`admin`	✗

`pool(`
`  tokens: Vec<Address>,`
`)`
		✓

`pools()`		✓
`get_two_pool_wasm_hash()`		✓
`get_three_pool_wasm_hash()`		✓
`get_admin()`		✓

B. Pool ABI

The Application Binary Interface (ABI) is defined by the generic procedural macro `generate_pool_contract`. Every public function extends the instance's time to live (TTL), except those that do not modify the state (the so-called "view" functions), `initialize()` and `upgrade()`.

Signature	Access control	Immutable

`initialize(`
`  admin: Address,`
`  a: u128,`
`  tokens: Vec<Address>,`
`  fee_share_bp: u128,`
`  admin_fee_share_bp: u128,`
`)`
	Only once	✗

`deposit(`
`  sender: Address,`
`  amounts: Vec<u128>,`
`  min_lp_amount: u128,`
`)`
	`sender`	✗

`withdraw(`
`  sender: Address,`
`  lp_amount: u128,`
`)`
	`sender`	✗

`swap(`
`  sender: Address,`
`  recipient: Address,`
`  amount_in: u128,`
`  receive_amount_min: u128,`
`  token_from: Token,`
`  token_to: Token,`
`)`
	`sender`	✗

`claim_rewards(`
`  sender: Address,`
`)`
	`sender`	✗

`claim_admin_fee()`	`admin`	✗

`set_admin(`
`  new_admin: Address,`
`)`
	`admin`	✗

`set_admin_fee_share(`
`  admin_fee_share_bp: u128,`
`)`
	`admin`	✗

`set_fee_share(`
`  fee_share_bp: u128,`
`)`
	`admin`	✗

	`admin`	✗

Signature	Access control	Immutable

`upgrade(`
`  new_wasm_hash: BytesN<32>,`
`)`		

`pending_rewards(`
`  user: Address,`
`)`
		✓

`get_pool()`		✓

`get_user_deposit(`
`  user: Address,`
`)`
		✓

`get_d()`		✓

`get_receive_amount(`
`  input: u128,`
`  token_from: Token,`
`  token_to: Token,`
`)`
		✓

`get_send_amount(`
`  output: u128,`
`  token_from: Token,`
`  token_to: Token,`
`)`
		✓

`get_withdraw_amount(`
`  lp_amount: u128,`
`)`
		✓

`get_deposit_amount(`
`  amounts: Vec<u128>,`
`)`
		✓

`get_admin()`		✓