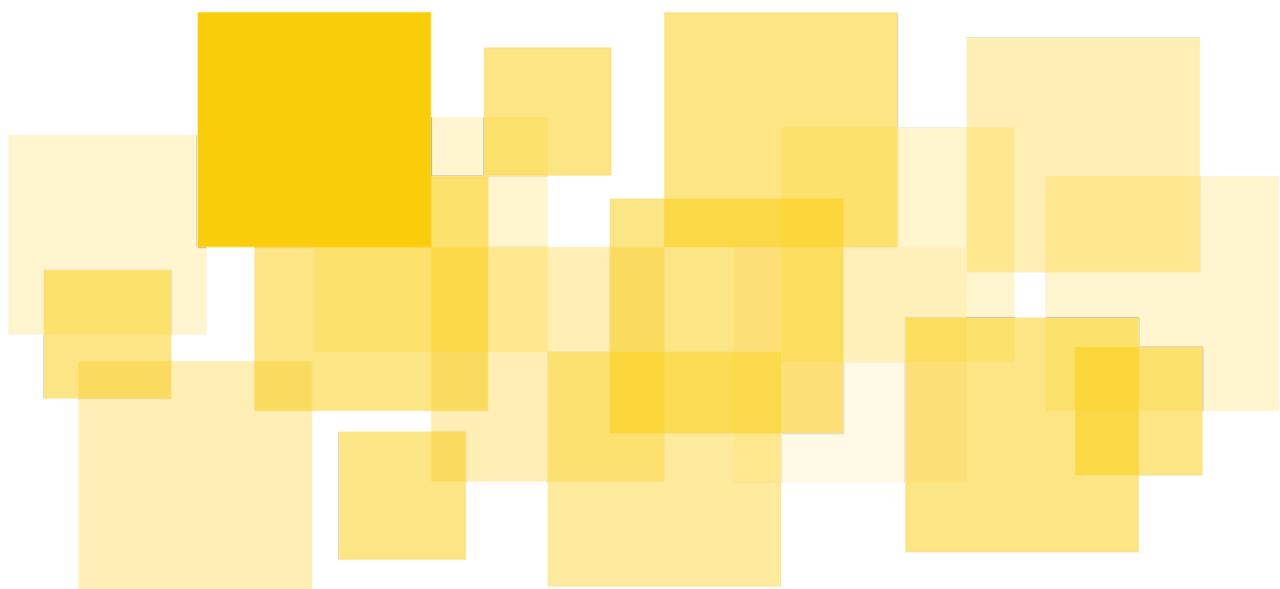




Security Audit Report

StellarBroker Stellar

Delivered: April 28, 2025



Prepared for SDF by





Table of Contents

- Disclaimer
- Executive Summary
- Goal
- Scope
- Methodology
- Platform Features and Logic Description
 - StellarBroker contract
- Invariants
- Findings
 - [A1] Swaps May Lead to the Accumulated Fees Drainage
 - [A2] The Protocol May Incorrectly Handle Fees
 - [A3] Parametric Fees Enable Users to Deny Profit For the Protocol
- Informative Findings
 - [B1] Best practices recommendations
 - [B2] The Fee Path May Lead to Arbitrary Assets



Disclaimer

This report does not constitute legal or investment advice. You understand and agree that this report relates to new and emerging technologies and that there are significant risks inherent in using such technologies that cannot be completely protected against. While this report has been prepared based on data and information that has been provided by you or is otherwise publicly available, there are likely additional unknown risks which otherwise exist. This report is also not comprehensive in scope, excluding a number of components critical to the correct operation of this system. This report is for informational purposes only and is provided on an "as-is" basis and you acknowledge and agree that you are making use of this report and the information contained herein at your own risk. The preparers of this report make no representations or warranties of any kind, either express or implied, regarding the information in or the use of this report and shall not be liable to you or any third parties for any acts or omissions undertaken by you or any third parties based on the information contained herein.

Smart contracts are still a nascent software arena, and their deployment and public offering carries substantial risk.

Finally, the possibility of human error in the manual review process is very real, and we recommend seeking multiple independent opinions on any claims which impact a large quantity of funds.

Executive Summary

StellarBroker engaged [Runtime Verification Inc.](#) to perform a security audit of its smart contract system as part of the Stellar Development Foundation's Audit Bank program. The audit was conducted between March 31 and April 18, 2025. The objective was to assess the security and correctness of the implementation, identify any exploitable vulnerabilities, and provide recommendations to enhance the system's reliability.

StellarBroker serves as a core component of the Soroban-based DeFi ecosystem, facilitating complex swaps and abstracting routing strategies to optimize the profits of users looking to trade in the Stellar ecosystem. Given its critical role and potential for a high volume of user engagement, ensuring the robustness and security of its smart contracts is paramount.

The audit process involved a comprehensive review of the smart contract codebase, focusing on both manual inspection and formal verification techniques. Runtime Verification utilized [Komet](#), its in-house formal verification and fuzz testing tool tailored for Soroban smart contracts, to rigorously test the system's behavior under various conditions. This included the development and analysis of key invariants to ensure the system's integrity across all state transitions.

The audit led to the identification of issues of potential severity for the protocol's health, which have been identified as follows:

- Potential threats to the contract's fund integrity: [\[A1\] Swaps May Lead to the Accumulated Fees Drainage](#), [\[A2\] The Protocol May Incorrectly Handle Fees](#), [\[A3\] Parametric Fees Enable Users to Deny Profit For the Protocol](#);
- Potential code/logic malfunction: [\[A2\] The Protocol May Incorrectly Handle Fees](#).

In addition, several informative findings and general recommendations have also been made, including:

- Best practices and code optimization-related particularities: [\[B1\] Best practices recommendations](#);
- Minor observations on the protocol's asset management: [\[B2\] The Fee Path May Lead to Arbitrary Assets](#).

All findings have been documented in the subsequent sections of this report, along with suggested mitigations to improve the security and maintainability of the StellarBroker system.



Any responses or fixes submitted by the client have been reviewed and noted as part of the report's finalization.

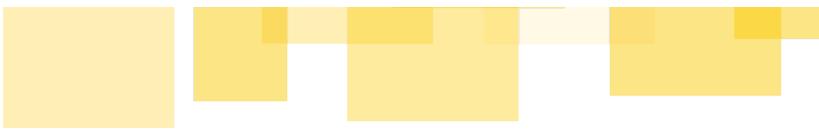


Goal

The goal of the audit is threefold:

- Review the high-level business logic (protocol design) of Stellar Broker's system based on the provided documentation and code;
- Review the low-level implementation of the system for the individual Soroban smart contract;
- Analyze the integration between abstractions of the modules interacting with the contract in the scope of the engagement and reason about possible exploitative corner cases.

The audit focuses on identifying issues in the system's logic and implementation that could potentially render the system vulnerable to attacks or cause it to malfunction. Furthermore, the audit highlights informative findings that could be used to improve the safety and efficiency of the implementation.



Scope

The scope of this audit is limited to the code contained in a single public GitHub repository provided by the StellarBroker team. The router contract was identified within this repository as the primary artifact under review for this engagement.

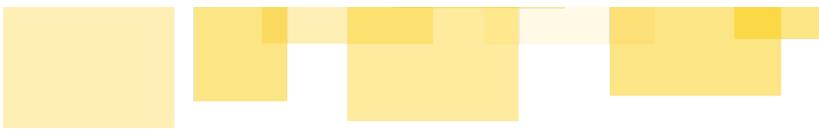
- Stellar Broker Router Repository (public)
 - <https://github.com/stellar-broker/router-contract>
 - Commit: `3a4f4dd47b0e679dee48be10a429fb749fc08c91`
 - `src/lib.rs` : Core contract responsible for orchestrating multi-source swaps across various liquidity venues on the Stellar network.

The Stellar Broker router acts as a unified interface to aggregate liquidity from multiple sources, including but not limited to SoroSwap, Aquarius, Phoenix, and Comet. It currently serves as the default swap router in the Albedo wallet, with additional integrations in progress.

The codebase under review consists of approximately 800 lines of Rust code written for the Soroban smart contract platform. In preparing for the audit, Runtime Verification referenced comments provided in the code, publicly available documentation, and supplemental materials shared by the Stellar Broker team.

The audit is strictly limited to the artifacts listed above. Off-chain components, frontend logic, deployment infrastructure, and third-party integrations are outside the scope of this engagement.

Commits addressing any findings presented in this report have also been reviewed to verify that identified issues were appropriately addressed prior to report finalization.



Methodology

Although manual code review cannot guarantee the discovery of all possible security vulnerabilities, as noted in our Disclaimer, we followed a structured and thorough approach to maximize the effectiveness of this audit engagement within the agreed timeframe.

The audit spanned three calendar weeks, with each phase of the process designed to identify and validate both high-level and low-level security concerns.

During the first week, we conducted a high-level design review of the StellarBroker router. We analyzed the architecture, key trust assumptions, and the security implications of interacting with various liquidity venues on the Stellar network. Particular emphasis was placed on identifying core invariants that should be upheld by the protocol under all valid inputs and runtime conditions.

Over the following two weeks, we thoroughly reviewed the contracts' source code to detect any unexpected (and possibly exploitable) behaviors. To facilitate our understanding of the platform's behavior, higher-level representations of the Rust codebase were created, including:

- Modeled sequences of logical operations, considering the limitations enforced by the identified invariants, checking if all desired properties hold for any possible input value;
- Manually built high-level function call maps, aiding the comprehension of the code and organization of the protocol's verification process;
- Performed fuzz testing with [Komet](#) to check if the specified invariants would hold for a set of randomized interactions with the contracts;
- Used static analyzers such as [Scout](#) to identify commonly identifiable issues;
- Created abstractions of the elements outside of the scope of this audit to build a complete picture of the protocol's business logic in action.

This approach enabled us to systematically check consistency between the logic and the provided Soroban Rust implementation of the system.

Finally, we conducted rounds of internal discussions with security experts over the code and platform design, aiming to verify possible exploitation vectors and identify improvements for the analyzed contracts. Code optimizations have also been discovered as an outcome of these research sessions and discussions.

Findings in this report stem from a combination of:

- 
- Manual inspection of the Rust source code.
 - Design-level reasoning about protocol behavior and system invariants.
 - Property-based testing and symbolic execution using Komet.

Throughout the audit, vulnerabilities and edge cases were reported to the StellarBroker team in real-time. This allowed for iterative discussion and clarification, accelerating the fix-and-verify cycle. In the final week of the audit, we compiled the report and validated any changes submitted by the client in response to previously communicated issues.

Additionally, given the nascent Stellar-Soroban development and auditing community, we reviewed [this list](#) of known Ethereum security vulnerabilities and attack vectors and checked whether they apply to the smart contracts and scripts; if they apply, we checked whether the code is vulnerable to them.

Platform Features and Logic Description

[StellarBroker](#) is a multi-source liquidity aggregator and swap router specifically designed for the Stellar network. It aims to solve the issue of fragmented liquidity across the various trading venues on Stellar, including the built-in decentralized exchange (DEX) and automated market makers (AMMs). By providing a unified interface, StellarBroker allows users to find the best prices and execute trades efficiently by routing orders through the optimal combination of available liquidity pools.

At the time of writing, StellarBroker uses Soroswap, Aquarius (stable swap and constant market maker formulas), Phoenix, and Comet protocols as liquidity sources.

The project achieves this by analyzing the depth and pricing across different exchanges and protocols in real-time. When a user initiates a swap, StellarBroker's smart routing algorithm determines the most cost-effective way to execute the trade, potentially splitting the order across multiple sources to get the best overall outcome. This not only improves price execution for users but also simplifies the trading experience by abstracting away the complexity of interacting with multiple individual platforms. For applications integrating with StellarBroker, it offers a reliable execution flow and a seamless API.

With the swap path(s) formulated using StellarBroker's interface or API, the smart contract in the scope of this engagement is invoked. It is tasked with executing the swaps according to the desired paths provided as parameters by the caller.

StellarBroker contract

The StellarBroker contract is responsible for enabling users to atomically execute a sequence of token swaps following one or more trade routes. It considers only two possible actors in its expected operations: the administrator, and normal users.

See the diagram in Figure 1 for the interface of interactions of the StellarBroker contract. Administrative functions are identified by their bold function name.

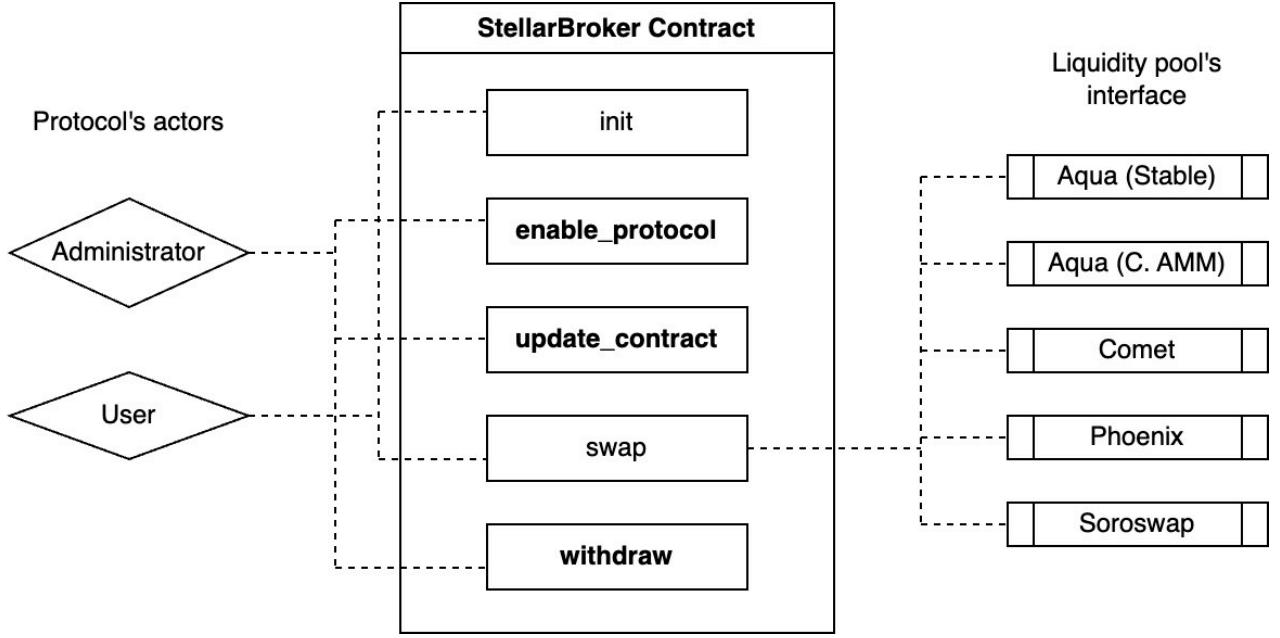


Figure 1: StellarBroker contract's interface and related actors.

The administrator is defined as the caller of the `init` contract function, which sets the instance storage entry for the administrative address (`ADMIN`) and extends the instance storage of the contract. `init` can only be called once, as it validates if the `ADMIN` storage entry is populated or not to prevent it from being overridden.

With the administrator set, the contract can now be updated using the `update_contract` endpoint, which validates the caller address against the `ADMIN` storage entry before updating the contract WASM. The other endpoint exclusive to the admin is the `enable_protocol` endpoint, which is used to either enable or disable a specific protocol based on an identification number ranging from 0 to 4. These numbers are used as identifiers for, respectively, the Aquarius Constant AMM, Aquarius StableSwap, Soroswap, Comet, and Phoenix protocols.

The remaining administrative function, `withdraw`, can be used by the contract administrator to withdraw any amount of tokens from the broker contract. It is intended to be used primarily for fee withdrawing but can be used in a scenario where assets are mistakenly sent to this smart contract.

Finally, the core functionality of the StellarBroker contract is contained in the `swap` function. It receives the following parameters:

- `selling` : indicates the address of the token that initiates all swap routes (i.e., the token being sold);
- `routes` : a vector containing a sequence of routes, where each route represents a trade path for a fraction of the total amount of tokens being sold;
- `trader` : address of the individual who would like to perform the swap (the caller);
- `vfee` : fee taken from the profit of the operation (1000 representing 100%);
- `ffee` : fixed fee over the total amount of tokens bought in the operation (1000 representing 100%);
- `fpath` : path of trades for converting the bought token into the fee token, in case the bought token has a different address than the fee token;

Although not enforced by the contract's logic, USDC is designated as the fee token according to the client's intended design. Practically, from a business logic perspective, the token used as the fee will not matter, as the administrator of the contract has the power to withdraw any token accumulated in the contract.

While `selling`, `trader`, `vfee`, and `ffee` are relatively straightforward parameters, `routes` and `fpath`, composite elements, can be broken down to better understand how the swaps are performed.

`routes` is a vector of a `struct` named `Route`. A route will represent a part of the total sequence of trades to obtain the buying asset. It is composed of an `amount` integer, which is how much of the selling asset will be traded in this route; a `min` integer, which represents the minimum accepted amount of the buying asset that the swaps in this route should return; an `estimated` integer, which represents the expected amount that the swaps in this route should return; and a `path`, which is a vector of `PathStep`.

`PathStep` is a structure that represents a single swap in a sequence of trades. It is composed of a `protocol` enumerator, representing the identifier of which protocol will be used to perform the swap; an `asset` address, which represents the asset being bought in that specific step; a `pool` address, representing the specific liquidity pool of the specified protocol in which the swap will be performed; and two integers `si` and `bi`, which work as helpers to identify the buying and selling assets being exchanged. A chain of `PathStep`s will determine a trading route starting on a pre-defined selling asset and ending on the desired buying asset.

To visualize all this, imagine that a user wants to perform a trade of 200 XLM to USDC. Using StellarBroker's interface or API, the user obtains the optimal trading route and is now ready to

call the broker smart contract. The `selling` address is XLM's token address, the `trader` address is his own address, `vfee` and `ffee` will be values defined by the platform based on how much will be charged from the operation's profit and swapped amount. The `routes` vector, with illustrative values, can be observed in Figure 2.

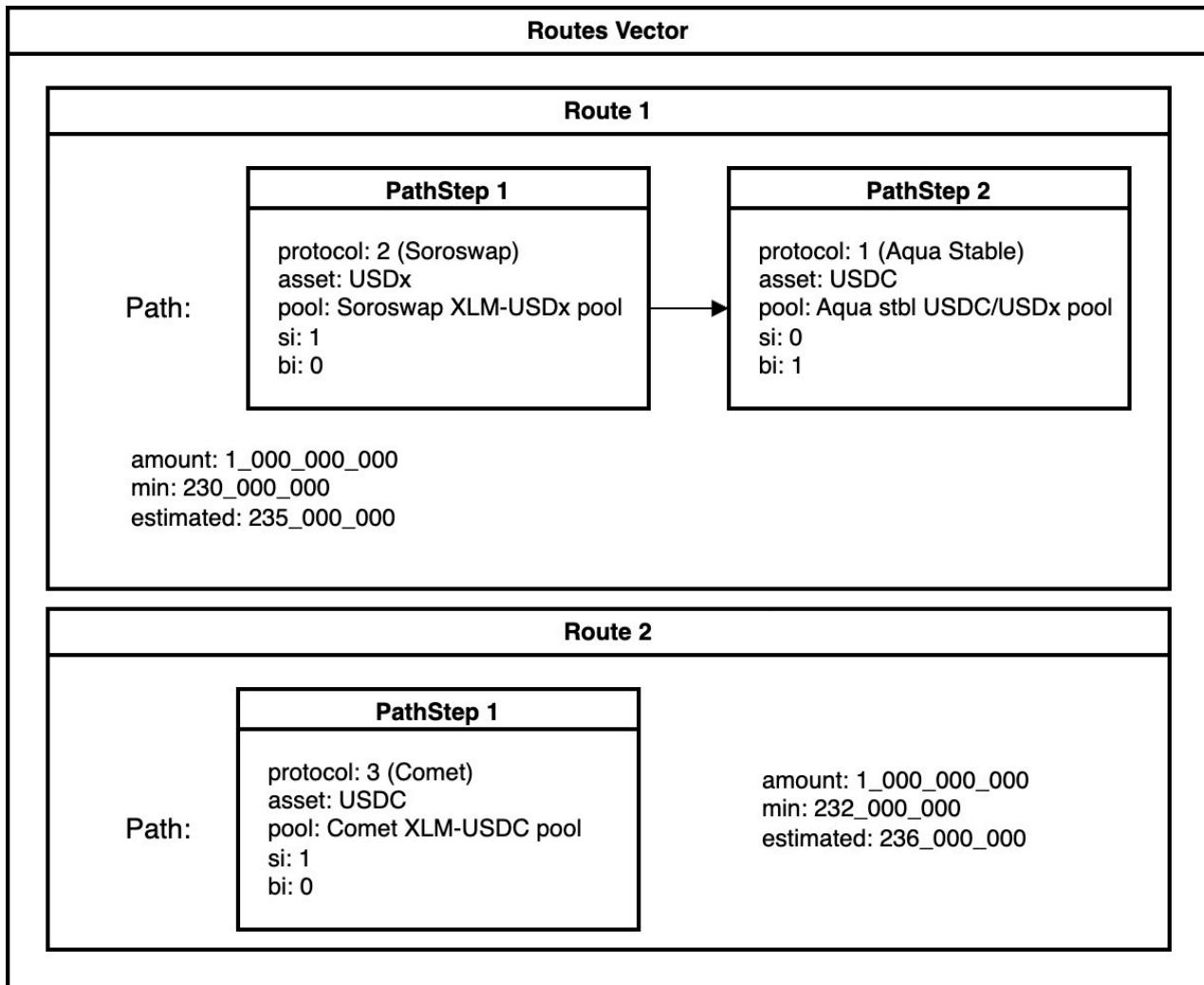


Figure 2: Sample Routes vector provided as a parameter for a swap operation (illustrative values).

Notice that, for this trade, the illustrative optimal route has been broken into two, each with half of the total value of the selling token being exchanged. One goes from XLM to USDx via the



Soroswap protocol and then converts the obtained USDx to USDC via an Aqua Stableswap pool, while a second route goes directly from XLM to USDC via the Comet protocol.

With the elements of the routes vector, it is possible to calculate the total amount of USDC bought, the minimum amount accepted of USDC (considering loss to slippage), and the actual estimated amount of USDC that should have been received. Furthermore, the actual returned value for each route is obtained when performing the swaps, and the sum of the bought token obtained will be used, together with the estimated amount that should have been bought, to calculate both the profit and the fees to be paid by this operation.

Considering that this operation's target token is USDC, and USDC is the fee token, `fpath` will be an empty vector as the fee is directly extracted from the bought amount. In a scenario where the buying or selling token is not the fee token, a vector of `PathStep`s must be provided, containing a trade path from the bought token to the fee token.

Invariants

During the audit, invariants were defined and used to guide part of our search for possible issues with StellarBroker's contract. Using the client's documentation, intended business logic, and references collected during the audit, we identified the following invariants:

- For any swapping operation performed, all token balances of the contract must either grow or stay the same;
- Users receive at least as much as the expected amount out for their swaps (according to the path's minimum bought amount);
- Token amount accrued as fees can only be withdrawn by the administrator;
- Each swap operation must result in exactly one asset leaving the caller's wallet and one asset being received, regardless of the number of routes used.
- Only enabled protocols can be used for swapping operations;
- Only the administrative figure can enable or disable protocols;
- It should not be possible to change the administrative address without updating the contract;
- Any transactions that disrespect the previous invariants should lead the contract to a panic state.



Findings

Findings presented in this section are issues that can cause the system to fail, malfunction, and/or be exploited, and should be properly addressed.

All findings have a severity level and an execution difficulty level, ranging from low to high, as well as categories in which it fits. For more information about the classifications of the findings, refer to our [Smart Contract Analysis](#) page (adaptations performed where applicable).

[A1] Swaps May Lead to the Accumulated Fees Drainage

Severity: Medium

Difficulty: Low

Recommended Action: Fix Design

Addressed by client

Description

As mentioned in the [Platform Features and Logic Description](#) section, a user that wishes to perform swaps through the StellarBroker contract must provide information about the assets being sold, as well as information about the routes taken to perform the swap, encoded in `PathStep` s.

When specifying each step of the trade route from the selling to the buying asset, the user also provides the address of the liquidity pool in which the assets will be exchanged. The aggravating detail here is that no validation over the address of this liquidity pool is performed, nor does the StellarBroker contract keep a reference to trusted liquidity pools. Therefore, when providing the address of the liquidity pool for a specific step, a malicious user may provide a contract address that implements the interface of that liquidity pool but, in practice, executes whatever code this third party wishes to.

It is important to highlight that, at the time of writing, the only exploitation identified and demonstrated was the drainage of tokens in the smart contract. Still, we note that this is achieved through executing code contained in an unverified, potentially user-provided smart contract. The depth of the consequences of this is not made immediately clear, given all the available functionalities that are and will be implemented in Stellar's ecosystem, as well as the unpredictability of complex systems when integrated with code of arbitrary content.

Scenario

The following is a sample exploit scenario that uses the abovementioned properties to drain assets held by the StellarBroker contract. It was implemented as a [Komet](#) test and provided to the client as a part of the audit deliverables.

Assume the broker contract has accumulated XLM tokens (e.g., from prior swaps as fees). An attacker performs a 2-step swap: USDC -> XLM -> EURC, and exploits the broker's failure to

verify actual token transfers at each step to drain XLM from the contract.

1. The attacker deploys a fake Comet contract that implements the real Comet interface.
 - It is initialized with the address of the final output token (EURC) and the amount of XLM to steal (denoted **N**).
 - The fake contract initially holds 1 EURC to satisfy the final output requirement.
2. The attacker constructs a 2-step swap route through the broker: USDC -> XLM -> EURC.
 - The fake Comet is used as the liquidity pool contract for both swaps.
 - The swap starts with 1 USDC.
3. Execution flow:
 - The broker receives 1 USDC from the attacker.
 - It calls the fake Comet to perform the first swap (USDC -> XLM).
 - The fake contract pulls 1 USDC from the broker but does **not** transfer any XLM.
 - Instead, it returns **N** as the simulated output amount.
 - Believing the swap succeeded, the broker proceeds to the second swap.
 - It calls the fake Comet to swap **N** XLM -> EURC.
 - The broker transfers **N** XLM (its own tokens) to the fake contract.
 - The fake contract sends back 1 EURC (from its initial balance).
 - The broker verifies only the final EURC output and considers the swap successful.
 - It completes the swap by transferring 1 EURC to the attacker.
4. Outcome:
 - The attacker receives 1 EURC in exchange for 1 USDC.
 - The broker has unknowingly lost **N** XLM to the attacker.
 - This is possible because intermediate token transfers are not validated—only return values are trusted.

Recommendation

Validate that liquidity pools being invoked are trusted. This can be achieved by managing the addresses of trusted pools on a map, validating if a provided pool address is present in the map

and is tagged as enabled.

Status

The client has addressed this finding by enforcing the correct progression of the fee balance at every swap. If a malicious third-party contract attempts to drain assets held in the broker, given that the broker should only hold a number of the fee assets, any attempt to drain contract funds will fail.

[A2] The Protocol May Incorrectly Handle Fees

Severity: Low

Difficulty: Low

Recommended Action: Fix Code

Addressed by client

Description

When performing a swap, the amount of assets bought, together with the fee percentages, are used to calculate the fees and subtract a fraction of the buying or selling token amounts. The balance from which the fee is deducted depends on whether the fee asset is the same as the buying token.

According to its initial implementation, if the user does not provide a fee path (`fpath`) when calling the swap function, or if the fee asset acquired from that fee path is the same as the buying asset, no modification to the value of the asset balances is performed, as the protocol will use zero as the value of the calculated received fees. Therefore, no fees will be charged from that swap, regardless of the percentage of fees that have been provided for charging the operation.

Recommendation

If a fee path is not provided, or if it is provided and the obtained asset from this path is the buying asset, deduct the fee amount from the buying asset balance.

Status

The client has addressed this finding by following the recommendation above.

[A3] Parametric Fees Enable Users to Deny Profit For the Protocol

Severity: Low

Difficulty: High

Recommended Action: Fix Design

Not addressed by client

Description

Fees collected from the swap operations executed through the StellarBroker contract define the protocol's revenue. These fees are calculated based on the amount of the swap operation, the profit obtained from using the StellarBroker, and fee percentages informed by the callers as parameters of the `swap` function.

Considering that the fees are parametric, a user who wishes to deny profit to the broker contract can provide zero as the value for the fees when calling the `swap` function.

It is important to highlight that a user cannot use StellarBroker's API to obtain a sample transaction and modify its parameters; the process of building and submitting the transaction is performed on the server side. Furthermore, doing this while using general transaction information inferred from any available interface for transaction signature would be considerably complex, considering the time taken to generate the transaction by the StellarBroker server.

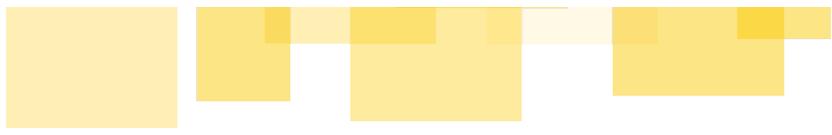
This finding could only be exploited if the user either finds a performative way to use a transaction generated by the StellarBroker server and modifies the fees parameters, or generates their own routing algorithm while still using StellarBroker's contract for executing the swaps. Both approaches are likely impractical due to the complexity and technical effort involved.

Recommendation

To prevent users from bypassing fee collection, consider enforcing fixed fee values within the contract logic, rather than accepting them as user-provided parameters.

Status

This finding has been acknowledged by the client.



Informative Findings

The findings presented in this section do not necessarily represent any flaw in the code itself. However, they indicate areas where the code may need external support or deviate from best practices. We have also included information on potential code size reductions and remarks on the operational perspective of the contract.

[B1] Best practices recommendations

Severity: Informative

Recommended Action: Fix Code

Partially addressed by client

Description

Here are some notes on the protocol's particularities, comments, and suggestions to improve the code or the business logic of the protocol in a best-practice sense. They do not present issues with the audited protocol themselves. Still, they are advised to either be aware of or to follow when possible, and they may explain minor unexpected behaviors in the deployed project.

1. There is a redundant check on the `update_contract` function. Using the `!e.is_initialized()` validation, the contract checks if the `ADMIN` storage is populated and fails if it isn't. Consecutively, it checks if the caller is the administrator. If the `ADMIN` storage hasn't been populated at this point, this second validation will fail regardless;
2. In the `swap` function, the fee asset can be fetched twice with the same results in lines 154 and 160;
3. When extending the contract instance storage Time-To-Live, the contract has a fixed extension time that can be triggered by any user, and this may cause arbitrary users to be tasked with the contract's TTL extension cost;
4. The release profile in `Cargo.toml` includes the `overflow-checks = true` flag, which enforces overflow and underflow checks in arithmetic operations. However, the codebase uses `checked_*` arithmetic operations followed by `unwrap()`, which reduces the code readability and defeats the purpose of using the compilation flag.

Recommendation

For each of the topics elaborated above, we recommend implementing the following approaches into the protocol's contracts:

1. Remove the `!e.is_initialized()` validation from the `update_contract` function;
2. Remove the redundant fetch of the fee asset in line 160;
3. Have different instance TTL extension thresholds for different actors, using a lower threshold for users and a higher threshold for the admin, preventing users from performing

the contracts instance storage TTL extension unless strictly necessary;

4. Use direct arithmetic operations instead of `checked_*` functions when the `overflow-checks = true` compiler flag is enabled, as it already enforces overflow checks.
-

Status

The client has partially addressed this finding by following the recommendations for topics 1, 2, and 3 discussed above. Topic 4 was acknowledged for the sake of unified code styling and future-proofing.

[B2] The Fee Path May Lead to Arbitrary Assets

Severity: Informative

Recommended Action: Fix Design

Addressed by client

Description

The purpose of the fee path (`fpath`) is to provide a way to convert the asset bought into an acceptable fee to the StellarBroker protocol, which, according to the client's design, is supposed to be the USDC token.

The fee path is obtained from a parametrized variable. While this simplifies the processing of the path to the StellarBroker contract, a user can provide arbitrary paths when performing swap operations, and these swaps may not necessarily end in the desired fee asset.

Considering that the administrator has the capability to withdraw any tokens held by the broker contract, this finding does not constitute a threat of any severity to the StellarBroker protocol.

Recommendation

When necessary, enforce that the fee path correctly leads from the buying token to the fee token.

Status

This finding has been addressed by the client with the modifications concerning finding [\[A1\] Swaps May Lead to the Accumulated Fees Drainage](#).