Security Audit Report

Soroswap Aggregator Stellar

Delivered: August 31, 2024

Prepared for PaltaLabs by

runtime verification

Table of Contents

Disclaimer

Executive Summary

PaltaLabs engaged Runtime Verification Inc. to conduct a security audit of the Soroswap Aggregator and Adapters contracts' code. The objective was to review the platform's business logic and implementation in Rust (Soroban) and identify any issues that could cause the system to malfunction or be exploited.

The audit was conducted over the course of 3 calendar weeks (July 15, 2024, through August 5, 2024) and focused on analyzing the security of the source code of Soroswap's Aggregator as well as the Adapters used to communicate with swap routing contracts from different protocols (Soroswap and Phoenix), which enables users to atomically perform optimal swaps using different protocols simultaneously. The swaps are split between multiple routers according to a custom pathing provided with the transaction, allowing the user to optimize the amounts returned by each protocol and enhancing the returns from that swap.

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

- Third-party code execution: A1, A3;
- Functional correctness: A2;
- Input validation: A2, B4;
- Potential threats to users' fund integrity: A1, A2, A3, B4.

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

- Best practices and code optimization-related particularities: B1, B2, B3;
- Notes on potential economic risks: B4;
- Blockchain-related particularities: B1, B5.

The client has addressed all of the potentially critical findings as well as the majority of the informative findings, and general recommendations have been incorporated into the platform's code. All of the remaining have been acknowledged by the client and deemed non-threatening to the integrity of the platform, when not intended by design.

Goal

The goal of the audit is threefold:

1. Review the high-level business logic (protocol design) of Soroswap's system based on the provided documentation and code;
2. Review the low-level implementation of the system for the individual Soroban smart contracts (Aggregator and Adapters);
3. Analyze the integration between the modules in the scope of the engagement and reason about possible exploitive 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 the audit is limited to the code contained in a public Github repository provided by the client (soroswap/aggregator) Within the repository and among other files, a few contracts are highlighted as in the scope of the protocol. The repository and contracts are described below:

1. Aggregator: this repository contains several contracts designed for the Soroswap Aggregator protocol. From it, four modules were audited:
    - Aggregator (`aggregator/contracts/aggregator` , commit id `55a95f267dca0826ee6c0291da9523f68dbcd4e2` , branch `main`): implements the core of the protocol, handling requests for distributed swaps by calling the responsible adapters with the necessary information for performing token exchanges;
    - Adapter Interface (`aggregator/contracts/adapters/interface` , commit id `55a95f267dca0826ee6c0291da9523f68dbcd4e2` , branch `main`): implements the `SoroswapAggregatorAdapterTrait` trait, which defines the standard endpoints for communicating with Adapters;
    - Soroswap Adapter (`aggregator/contracts/adapters/soroswap` , commit id `55a95f267dca0826ee6c0291da9523f68dbcd4e2` , branch `main`): provides the functions for the aggregator trait enabling the protocol to communicate with the Soroswap router;
    - Phoenix Adapter (`aggregator/contracts/adapters/phoenix` , commit id `ed4ce017838ddd4f5c73a8666241c47b55ee1828` , branch `main`): provides the functions for the aggregator trait enabling the protocol to communicate with the Phoenix router;

Regarding implementations and modifications related to the findings raised in this audit engagement, the last analyzed state of the Soroswap Aggregator repository was on commit id `f149fc92206808d94b1909732f9577abe82a9fa0` , which encompasses a merge request to the `main` branch.

The comments provided in the code, a general description of the project, including samples of tests used for interacting with the platform, and online documentation provided by the client were used as reference material.

The audit is limited in scope to the artifacts listed above. Off-chain, auto-generated, or client-side portions of the codebase, as well as deployment and upgrade scripts, are not in the scope of this engagement.

Commits addressing the findings presented in this report have also been analyzed to ensure the resolution of potential issues in the protocol.

Methodology

Although the manual code review cannot guarantee to find all possible security vulnerabilities as mentioned in our Disclaimer, we have followed the approaches described below to make our audit as thorough as possible.

First, we rigorously reasoned about the business logic of the code, validating security-critical properties to ensure the absence of loopholes in the business logic. To this end, we carefully analyzed all the proposed features of the platform and the actors involved in the lifetime of deployed instances of the audited contracts.

Second, 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, where the most comprehensive were:

- Modeled sequences of mathematical operations as equations and, 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;
- 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.

Furthermore, once the creation of higher-level abstractions and the process of understanding contracts was complete, attempts to use the protocol's business logic to break the collected invariants were performed, resulting in some of the findings presented in this report.

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

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 Logic and Features Description

The Soroswap Aggregator, Adapter Interface, and Adapters (Soroswap and Phoenix) compose a protocol built using the Soroban platform in the Stellar network. At the moment of writing this report, the routers used by the adapters, the liquidity pools, and pair contracts used to swap tokens have already been deployed, and a documentation page is available for users with a high-level description of the overall protocol parts and business logic. This and other information about the protocol can be accessed through Soroswap's landing page.

The following is the general description of each contract in the scope of this audit engagement.

Aggregator

The Aggregator contract allows users to request the exchange of assets using several protocols as sources of liquidity at the same time. When initializing the Aggregator, the caller will automatically become the administrator of that protocol, set the contract as initialized, and a set of supported protocols provided as a parameter are registered in the contract's instance storage. The set of operations that the Aggregator undergoes when being initialized is seen in Figure 1.

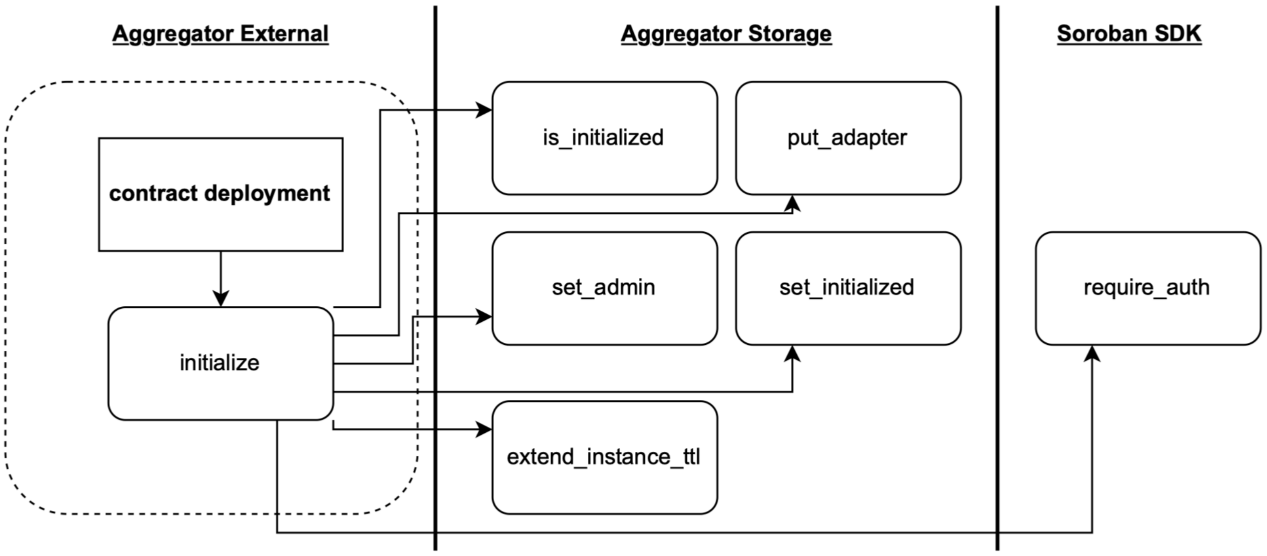

Figure 1: Aggregator "initialize" function operations diagram.

To achieve its goal, the Aggregator interface allows users to call for these simultaneous swaps using the `swap_exact_tokens_for_tokens` and `swap_tokens_for_exact_tokens`, where the user informs the input and output token for the swaps as well as the amounts involved in the operation and the deadline in which it must be accepted into the chain. Additionally, when calling these functions, the user must submit the distribution of swaps among protocols in a vector of `DexDistribution` objects, which are responsible for specifying the protocols in which each part of the token exchange must be performed, the fractions of the total amount of tokens that must be exchanged, and the path for that exchange (sequence of tokens from the input to output). In Figure 2, we can have a glimpse of the function calls performed by the Aggregator in order to forward the swap requests to an Adapter.

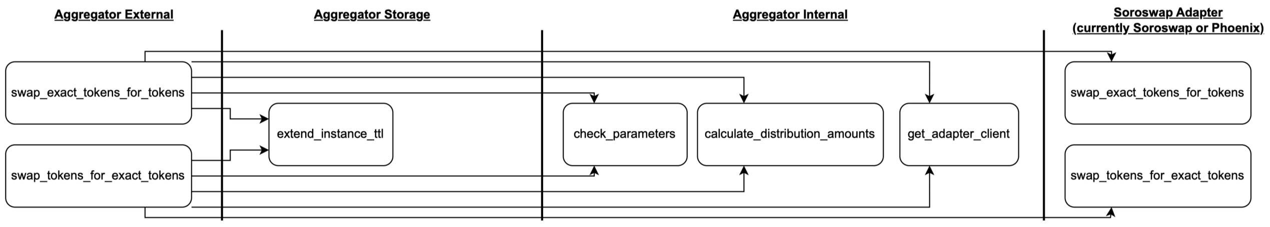
Aggregator External
Aggregator Storage
Aggregator Internal
Soroswap Adapter
(currently Soroswap or Phoenix)
swap_exact_tokens_for_tokens
swap_tokens_for_exact_tokens
extend_instance_ttl
check_parameters
calculate_distribution_amounts
get_adapter_client
swap_exact_tokens_for_tokens
swap_tokens_for_exact_tokens

Figure 2: Aggregator swap functions operations diagram.

Based on the total amount to be exchanged and the parts specified within the `DexDistribution` object, the individual amounts to be swapped at each protocol are defined in `calculate_distribution_amounts`. First, the total number of parts used to represent 100% of the value to be exchanged is obtained, and this is done by iterating over the distribution vector and adding the parts field. The result of this sum will be referred to as $total\_parts$. In a distribution vector($dist$) of size n, a particular distribution is referred to by $dist_i$ where i is the position of distribution in vector, and the amounts to be exchanged in the protocols at the positions 1 to $n-1$ are defined as:

$$amount_i = \lfloor \frac{total\_amount \times dist_i.parts}{total\_parts} \rfloor ; \forall i \in \mathbb{Z} \ s.t. \ 1 \leq i < n$$

For the last distribution in $dist$, the amount to be exchanged in that platform is defined as

$$amount_n = total\_to\_swap - \sum_{i=1}^{n-1} amount_i ; \ i \in \mathbb{Z}$$

Once the amounts to be exchanged in each platform are defined, the swap requests are forwarded to the Adapters using identifiers for the protocols available in the `DexDistribution` objects.

Besides having the capability to process swap requests by users, the Aggregator contract has a set of administrative functions that allow administrators to initialize the contract, register, pause, unpause, and remove protocols as adapters, modify the admin `address`, and upgrade the contract.

Adapter Interface

Defined in the `AdapterTrait` (formerly named `SoroswapAggregatorAdapterTrait` in the frozen repository version at the beginning of the engagement) is the interface for communication between the Aggregator and the Adapter. By implementing Adapters from this trait, the developers can have a standard way to call for swaps in different protocols, which require different methods and data to communicate with swap routers, but still use the method call for the Aggregator.

Adapters must implement the following functions:

- `initialize`: sets the necessary information for the functioning of the Adapter (protocol id, address, and initialized flag);
- `get_protocol_id`, `get_protocol_address`: getters for the Adapter data;
- `swap_exact_tokens_for_tokens`, `swap_tokens_for_exact_tokens`: methods called by the Aggregator that enable calls for protocol swap routers specifying either a minimum amount as input or output.

Adapters (Soroswap and Phoenix)

The implementations of the `AdapterTrait` are intended to handle the communication between the Aggregator and individual protocol swap routers for which each adapter is responsible. Included in the scope of this audit are the only currently developed Adapters for the Soroswap and Phoenix routers.

For both adapters, the functions responsible for initialization and data fetching are the same. The implementation of the `swap_exact_tokens_for_tokens` and `swap_tokens_for_exact_tokens` differ, though. Given that each protocol follows a different standard for performing swap operations, the preparation of the parameters is different according to which contract is being called.

The Soroswap swap router follows the same standard as the Aggregator, simplifying the implementation of the Soroswap Adapter to a contract that forwards the parameters for the swap request from the Aggregator to the router directly.

The implementation of the Phoenix Adapter is slightly more complex since adaptations to the parameters need to be performed to comply with the data structures used by the Phoenix contracts. These preparations range from the creation of new objects to represent the path of a swap to the reversing of the path for the `protocol_swap_tokens_for_exact_tokens` operation.

Having a simpler interface when compared to the Aggregator, the operations performed by the adapters post initialization can be seen in the diagram shown in Figure 3.

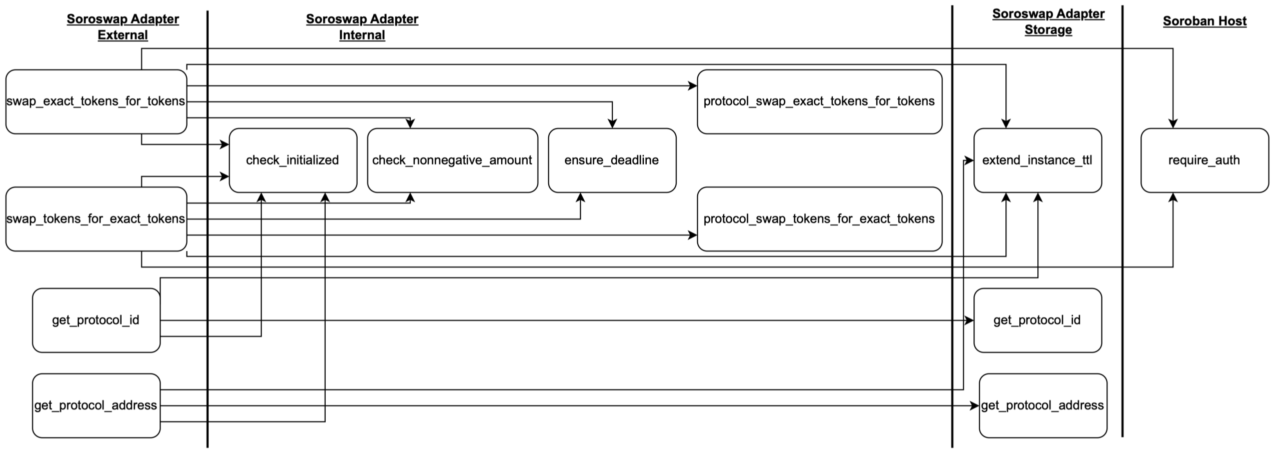

Figure 3: Soroswap Adapter swap functions and available data fetchers diagram.

Invariants

During the audit, invariants have been defined and used to guide part of our search for possible issues with both Aggregator and Adapter contracts. With the help of the Soroswap team, the following invariants were gathered:

Aggregator

- Contract privileges:
  - There are only two types of privileges on the Aggregator:
    - Admin: can call the Aggregator's administrative functions;
    - Users (not Admin): can request swaps.

- Storage considerations:
  - It is not possible to have two equal elements in the list of protocols stored in the Aggregator (`DataKey::ProtocolList`);
  - It is not possible to have two equal elements in the list of Adapters stored in the Aggregator

- The Aggregator does not have an administrator until `initialize` is called;
- The Aggregator always has an administrator after `initialize` is called;
- Only Admin can transfer administrator rights to another address;
- When performing swaps:
  - Users cannot attempt to trigger swaps of 0 asset amount; (`DataKey::Adapter(protocol_id)`).
  - When calling `swap_exact_tokens_for_tokens` , the user balance for the output token after the swap must increase by a value greater or equal to the minimum amount specified for the output;
  - When calling `swap_tokens_for_exact_tokens` , the user balance for the input token after the swap must decrease by a value smaller than the maximum amount for the input token specified;
  - The length of amounts to be swapped in each protocol list must have the same size as the vector of distributions provided by the user.

Adapters

- Privileges
  - Only 1 privilege level and it is the general user;
- Storage
  - After initialization, `DataKey::ProtocolId` and `DataKey::ProtocolAddress` are permanently set as there are no external functions to rewrite or delete these stored values;
- When paused by the Aggregator admin, it should not be possible to use it.

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] Adapters Can Be Hijacked By Third-Parties

Severity: Medium | Difficulty: Medium | Recommended Action: Fix Design | Addressed by client

Description

Within the functioning logic of the Aggregator and its Adapters, a single Aggregator is first deployed, initialized, and used to register new Adapters so that swap requests may be forwarded. To register these contracts, they must be deployed in the blockchain but no validations are enforcing that an Adapter must be initialized prior to being registered on the Aggregator. Furthermore, the process of completely setting up an Adapter is a two-step operation: first deploying the contract, and then calling the `initialize` function to set the protocol information, including the router contract address.

At the time of writing, Stellar-Soroban does not allow constructors into their contracts, meaning that the deployment and initialization function calls will be submitted to the blockchain as two different transactions. While this would not be an issue to the Aggregator itself, as it only has to be deployed once, ramifications for the Adapter may exist. This opens some windows to improbable, but dangerous scenarios where a malicious user monitors the admin or the deployer of Adapters and acts on top of submitted transactions.

The scenarios of severity are the ones where a malicious user attempts to front-run the administrator after deploying an Adapter, effectively calling the `initialize` function to give his own contract address instead of a correct protocol swap router address intended by the admin. Since adapters cannot have their data directly modified (as no methods for doing so are available), administrators are required to either not register this Adapter in the Aggregator and create a new one or, if already registered in the Aggregator, update it with a non-compromised adapter as quickly as possible, providing a brief window of time where the Aggregator may become a vector for execution of unknown third-party code.

Scenario

To clarify this finding, the following scenarios were elaborated to demonstrate how the presented issues can be exploited:

Execution of third-party code

1. The Aggregator's administrator (Admin) deploys a new Adapter;
2. Admin registers the newly deployed Adapter in the Aggregator, and has not initialized it yet;
3. A malicious user that noticed Adapters being deployed by the same address monitors the chain looking for new deployments, noticing the instantiation of this new Adapter;
4. The malicious user, immediately after the Adapter's creation, attempts to call `initialize`, giving a custom address for the protocol;
5. For the duration of time that it would take for Admin to overwrite the compromised Adapter, it would be possible for the Aggregator to perform function calls to unknown third-party code.

Griefing attack

1. Assume that the Admin only registers initialized and trusted adapters;
2. A malicious user that noticed Adapters being deployed by the same address monitors the chain looking for new deployments, noticing the instantiation of this new Adapter;
3. The malicious user, immediately after the Adapter's creation, calls `initialize` and compromises the Adapter;
4. Admin notices it and now has to create a new Adapter;
5. While creating Adapters from the Admin address, a griefing user may continue trying to effectively deny Admin of the capability of deploying Adapters, forcing other addresses to be used in the deployment of these contracts.

Recommendation

We highly recommend that the deployment and initialization of the Adapters happen programmatically through a deployer smart contract (https://soroban.stellar.org/docs/tutorials/deployer). This way, as we wait for constructors to be introduced to Soroban, the Soroswap Aggregator itself or its team will have a way to deploy and initialize the Adapters atomically and prevent scenarios similar to the above from happening.

Status

This finding has been addressed by the client. As presented in PR #67 of Soroswap Aggregator's repository, the client demonstrates how the deployment of new adapters is done using a deployer contract.

[A2] Malformed Paths on Swap Requests Are Allowed by the Aggregator

Severity: Low | Difficulty: Low | Recommended Action: Fix Code | Partially addressed by client

Description

When submitting a swap request to the Aggregator, among the data submitted about how the swap should be split among protocols is the path of exchanges. The path of a swap indicates which tokens should be used to obtain the desired token given the token provided as input. One example of a path would on a hypothetical transaction to exchange XLM for ETH where no pair contract in that specific protocol with enough liquidity would be available to perform such an exchange, so alternatively the user decides to exchange XLM for USD, and USD for ETH. A path for such an operation would be `[XLM, USD, ETH]`.

For the Soroswap Aggregator, all responsibility of forming the paths for the distribution is currently delegated to the caller or the entity that builds the transactions calling the Aggregator (i.e., the front-end of Soroswap Aggregator, the user himself, or a third-party that calls the Aggregator for the user). This means that paths for distribution allow, for instance:

1. Swap sequences that start and end in the same asset;
2. Different distributions with different starting and ending assets in their paths (e.g., exchanging XLM for ETH in Soroswap and XLM for USD in Phoenix);
3. Repeated sequences in a single path (i.e., token A -> token B -> token A -> token B);

While it is understandable that the official front-end of the Soroswap will correctly build the transactions and that misbehaviors are unlikely to happen, eventual updates or potential bugs in its path-finding algorithm may serve malformed paths in transactions to users. More likely are also the cases where a user trying to interact with the contract using self-developed scripts, which can submit transactions to the aggregator contract causing potential losses to this user.

Furthermore, the absence of these checks may cause behaviors that do not comply with the invariants for the Aggregator, which essentially compose the definition of what an Aggregator is supposed to provide for its users.

Recommendation

To prevent users from incurring losses due to interactions with the Aggregator, it is important to validate that the distribution paths respect rules regarding their integrity, for instance, enforcing that the paths start and end in the assets specified as input and output for the swap functions and that a path cannot start and end in the same token.

Although this is not mandatory, handling scenarios like the ones above can prevent corner cases from causing a bad user experience on the Soroswap platform. It is also needed to take into consideration which (if any) scenarios should be handled, as computationally intensive implementations could impact the price of transactions.

Status

This finding has been partially addressed by the client by enforcing that all distributions' paths start and end with the same token defined as `token_in` and `token_out`, respectively. Other possibly malformed paths have not been addressed due to scenarios where they are purposefully built by the users for their rightful advantage, such as "repeated sequences in a single path" (elaborated above), which has not been addressed so the Soroswap Aggregator can be used as a tool for performing triangular arbitrage.

More information about how the client addressed this issue can be found in PR #71 of the Soroswap Aggregator repository.

[A3] Subcontracts called from Aggregator have potential to call into malicious code

Severity: High   Difficulty: High   Recommended Action: Document Prominently
Addressed by client

Description

The nature of the Aggregator protocol is that it transitively calls multiple sub-contracts first through the Adapter contracts, and then to the exchanges themselves. As long as the signature of the transaction passes the `require_auth` checks at each level, the called contracts have full authority to manipulate the signer's funds. This coupled with the ability for contracts to upgrade means that there is always a chance that a protocol the Aggregator accesses could be upgraded to malicious code. A call to the Aggregator's swap functions that reaches malicious code has the potential to drain some or all of the signer's funds so long as they pass the min or max check on the dynamic asset (e.g. swap_exact_tokens_for_tokens).

Recommendation

The opportunity for subcontract calls to be execute malicious code replaced through an upgrade is a possibility with any contract that implements a subcontract call, and is in no way unique to this protocol. Currently, there is no way to restrict the assets accessed by the called contract to a restricted set (at least that we are familiar with). A whitelist of approved tokens is likely intractable as the malicious code could transfer *any* other assets, meaning that the checks need to be performed on the set of assets that are **not** either `asset_in` or `asset_out`. The best course of action to mitigate the risk of such a vulnerability is to carefully inspect and understand the footprint and auth payloads of each contract call of the transaction. However, it should be noted that not every user will have sufficient understanding of this information to be able to discern when a transaction will have malicious effects. Documentation for the Aggregator should notify users of this risk, and link them to relevant information and educational material to help them avoid exploitation.

Status

This finding has been acknowledged by the client. The modifications to the documentation highlighting the aforementioned issues have been added to the developer documentation of Soroswap Aggregator, as seen referenced in issue #65 of the Aggregator repository.

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 and Notable Particularities

Severity: Informative Recommended Action: Fix Code Partially addressed by client

Description

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

1. Aggregator storage function `remove_adapter_id` is somewhat incorrectly named, to match the other functions it should be `remove_protocol_id`;
2. Aggregator function `calculate_distribution_amounts` calls `Iterator::sum` which allows overflows for production code. However, since `overflow-checks=true` is set in the workspace `Cargo.toml`, this should not actually create an overflow. Still, it is recommended that even with this set that code should be defensive of overflow;
3. Specific functions are repeated throughout the `aggregator` repository, such as `check_nonnegative_amount`, `ensure_deadline`, and `check_initialized`. Having multiple instances of the same function, aside from lengthening the code base, may also create issues if needed to be modified, potentially creating different instances of functions that were supposed to do the same task. Furthermore, the instances of `check_nonnegative_amount` have the `pub` function modifier, but there is no need for it;
4. In the interface module, a trait is defined for specifying the functions that should be available in all adapters available for the protocol. This trait is named `SoroswapAggregatorAdapterTrait`, and its nomenclature doesn't reflect exactly what its purpose is, as it may remit to an adapter to the Soroswap Aggregator itself;
5. Aggregator external function `initialize` does not call `check_initialized` helper function and calls `is_initialized` storage function directly. Not only is this resulted in two different functions of similar purpose, but it may also be confusing for developers when modifying one of the functions, potentially expecting (by a mixup) that the changes should be reflected on the other;

6. The Aggregator can control the access to the Adapters by setting them as paused or not. Still, an Adapter can be directly called and used by users if called directly, instead of going through the Aggregator.

Recommendations

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

1. Use function names that correctly reflect the purpose of the operations they perform;
2. Validate the overflow possibility into the smart contract code;
3. Avoid duplicates in the codebase by creating a shared module among the contracts, and if a function isn't used outside of the module in which it is defined, do not declare it as public;
4. We recommend changing it to a more precise and simple name such as `AdapterTrait` to improve code readability;
5. Functions should exist for exclusive purposes, so avoid having two or more different functions with similar purposes to one another;
6. Move the `paused` property to the Adapter and validate if they are paused within the Adapter itself.

Status

All of the items raised in this finding, with the exception of item number 6, have been addressed by the client. References to the code merging operations addressing them, as well as the justification for not addressing item number 6 can be found in issue #66 of the Soroswap Aggregator repository.

[B2] Administrative Functions Have Redundant Validations

Severity: Informative   Recommended Action: Fix Code   Addressed by client

Description

On the Aggregator's administrative functions, all operations start by performing a set of checks mainly comprised of two function calls: `check_initialized(&e)?;` and `check_admin(&e)?;`. The first one is not necessary, except for the `set_admin` function.

An aggregator contract in its current state can only have an admin if `set_admin` is called or if the contract is initialized. Once an admin is set in the contract storage, it cannot be removed, only updated. This means that, given that the Aggregator's administrative functions first validate if the contract is initialized, and then check if an admin address has been set in the contract's storage before checking if the caller is indeed the protocol's administrator, the second check will always return true if the first one is true, making it redundant. The only exception to this is the `set_admin` function, which ideally should not be called before the contract initialization.

Recommendation

On all administrative functions of the Aggregator, except for `set_admin`, use only the `check_admin` for validating if the contract has been initialized and if the caller is indeed the administrator.

Status

This finding has been addressed by the client in PR #69 of the Soroswap Aggregator repository.

[B3] Redundant Validations in the Aggregator and Underlying Contracts

Severity: Informative · Recommended Action: Fix Code · Addressed by client

Description

In its current state, transactions requesting swaps to the Aggregator will have their parameters validated at a minimum of two times. When performing token exchanges, there are parameter checks in the Aggregator, the Adapters, and, in the case of the Soroswap router contract, in the router itself (all executed with `check_parameters`).

Similarly, the Aggregator contract validates that the parts of all distributions are greater than zero, meaning that all protocols used in the swap must exchange a value greater than 0, but once again in the Soroswap pair contract, a check preventing such operation from happening is present, making this validation redundant.

Recommendation

While smart contracts should be self-sufficient regarding validations, the Aggregator contract will perform several consecutive contract calls, reflecting potentially expensive transactions. Given the joint operation of these contracts, validations regarding the parameters and swap values should be performed at an Adapter level, if needed, and validations that **are confirmed present** at the router or underlying pair contracts can be made absent for the sake of the cost of transactions.

Status

This finding has been addressed by the client in PR #68 of the Soroswap Aggregator repository.

[B4] The Aggregator May Try to Trade Negligible Amounts

Severity: Informative | Recommended Action: Fix Design | Addressed by client

Description

On the calculation of the last amount added to the list of `swap_amounts` (in lines 104 to 112 of the Aggregator contract), the amount to be traded in a specific protocol can be negligible, leading to possible denial of transactions and weakening the purpose of validating that the parts of the distribution are different from 0 (line 83 of the Aggregator contract).

When calculating how much should be exchanged in each platform, all but the last value are calculated as `amount = (total_amount * part)/total_parts`. If the ratio of `part` to `total_parts` is too low, so will the amount.

Similarly, for the last element of the distributions, since all `part`s are greater than 0 and `total_parts` is the sum of all parts, `part` will never be equal to `total_parts`, meaning that there will always be a remaining value to be exchanged in the last distribution. This means that even though the parts for the last distribution will not effectively be used in calculating the amount to trade, the amount traded at the last distribution will always be greater or equal to 1.

What may emerge as a complication of this is how the AMMs will handle such low amounts. Two possibilities come to mind: (1) the transaction is refused as the input amount is too low, and (2) loss by division in the calculation of amounts to be exchanged may yield no return for the user.

Recommendation

As in finding A2, this may be a consequence of how the transaction and its parameters are built (either by the user or the front end), but an approach to avoid such scenarios is to, instead of validating if the `distribution.part` is not zero, validate that the amount to be exchanged is above a minimum amount.

Status

This finding has been addressed by the client in PR #67 and later updated in PR #76 of the Soroswap Aggregator repository. A minimum of 10 tokens has been established for performing swaps in each distribution.

[B5] The Aggregator Stores Dynamically Sized Data in Instance Storage

Severity: Informative Recommended Action: Fix Code Not addressed by client

Description

As a general rule, dynamically stored data should not be stored in Instance storage and should be stored in Persistent storage instead. Two reasons that Persistent storage is recommended over Instance storage are:

1. All Instance storage is ready for any call to the contract, even if the dynamic data is not accessed;
2. There becomes a risk of overflowing the allocated buffer for Instance storage as the dynamic data grows in size.

The functions responsible for storing and fetching the protocol ids in the aggregator (lines 72 to 87 of the storage module of the Aggregator) specifically use Instance storage to handle a `Vec<String>`. This vector holds the identifiers for the protocols that can be used to swap assets. It is expected that the number of identifiers stored in this vector is less than 10, and so this should cause negligible overhead for reason number 1, and there should be no risk of issues related to reason number 2 (listed above). That being said, there are no guards in the contract itself to protect from the vector growing unbounded, and so this is only enforced by the Admin, and without consideration, it is possible to grow the vector to an inappropriate size.

Recommendation

When handling data of dynamic size in storage, use Persistent storage instead of Instance.

Status

This finding has been acknowledged by the client. The reasoning for the acknowledgment can be found elaborate in issue #52 of the Soroswap Aggregator repository.

Appendix A: Soroswap Aggregator and Adapter diagrams

Aggregator Storage: 1

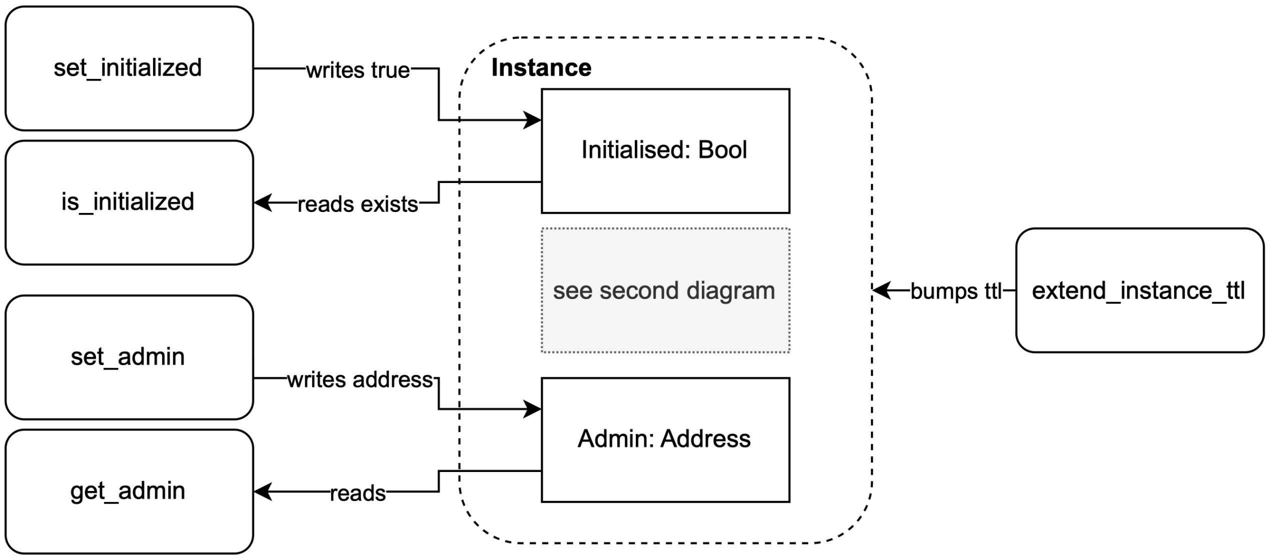

Figure 1: Aggregator functions in relationship to storage usage (1).

Aggregator Storage: 2

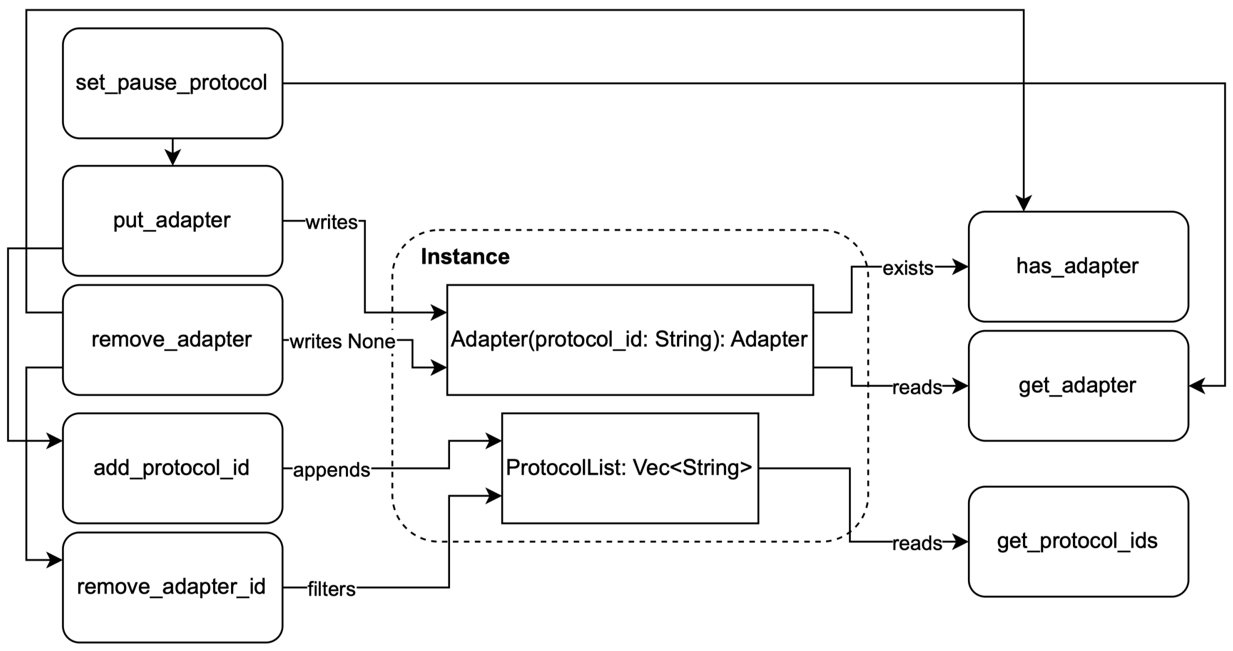

Figure 2: Aggregator functions in relationship to storage usage (2).

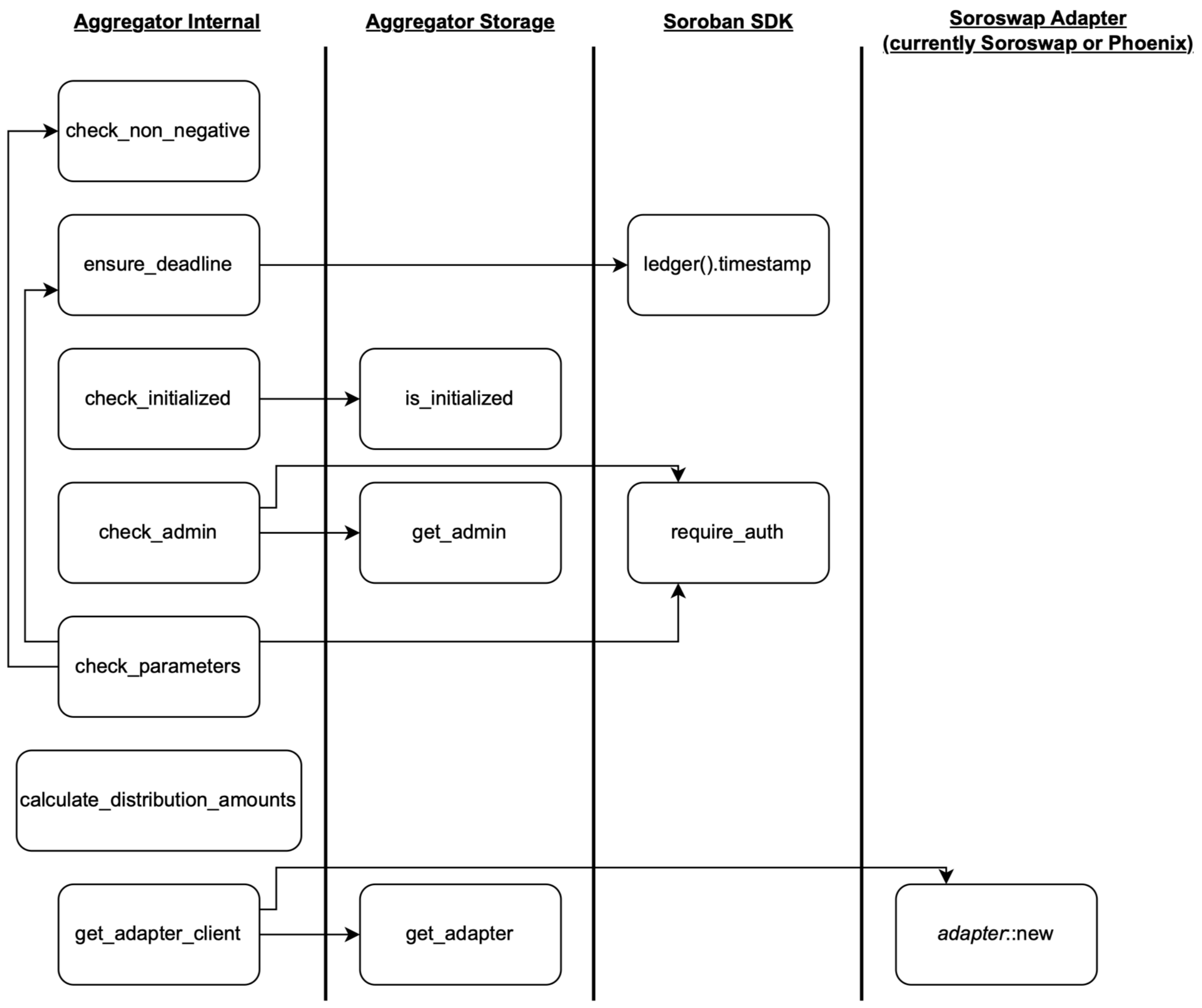

Figure 3: Aggregator function diagram (1).

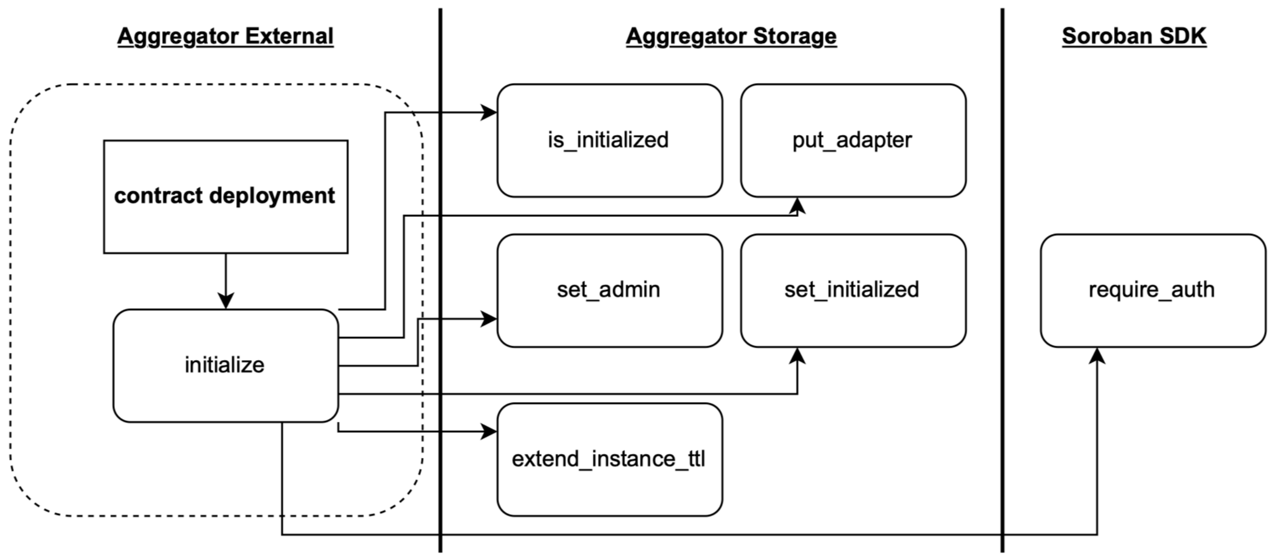

Figure 4: Aggregator initialization.

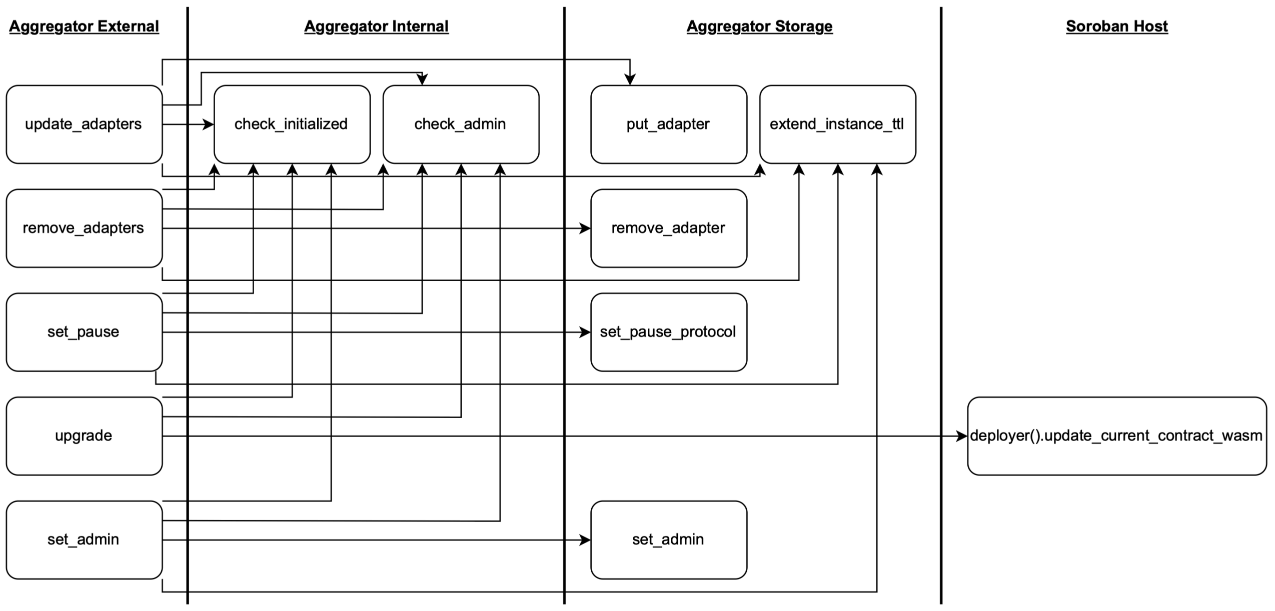

Figure 5: Aggregator function diagram (2).

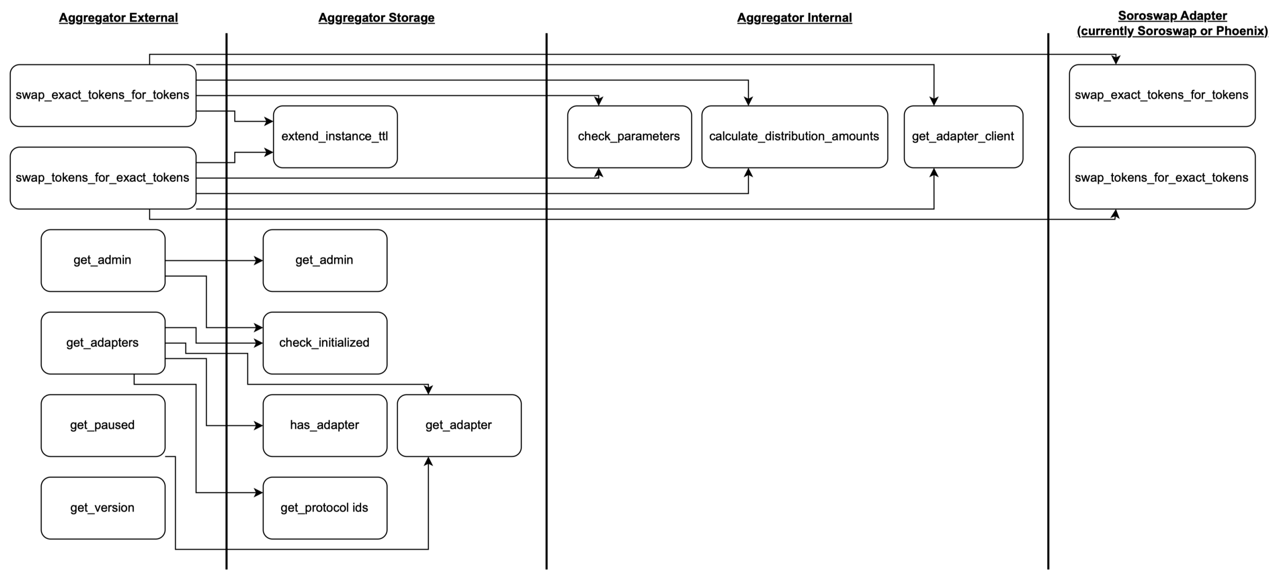

Figure 6: Aggregator function diagram (3).

<u>Soroswap Adapter Storage</u>

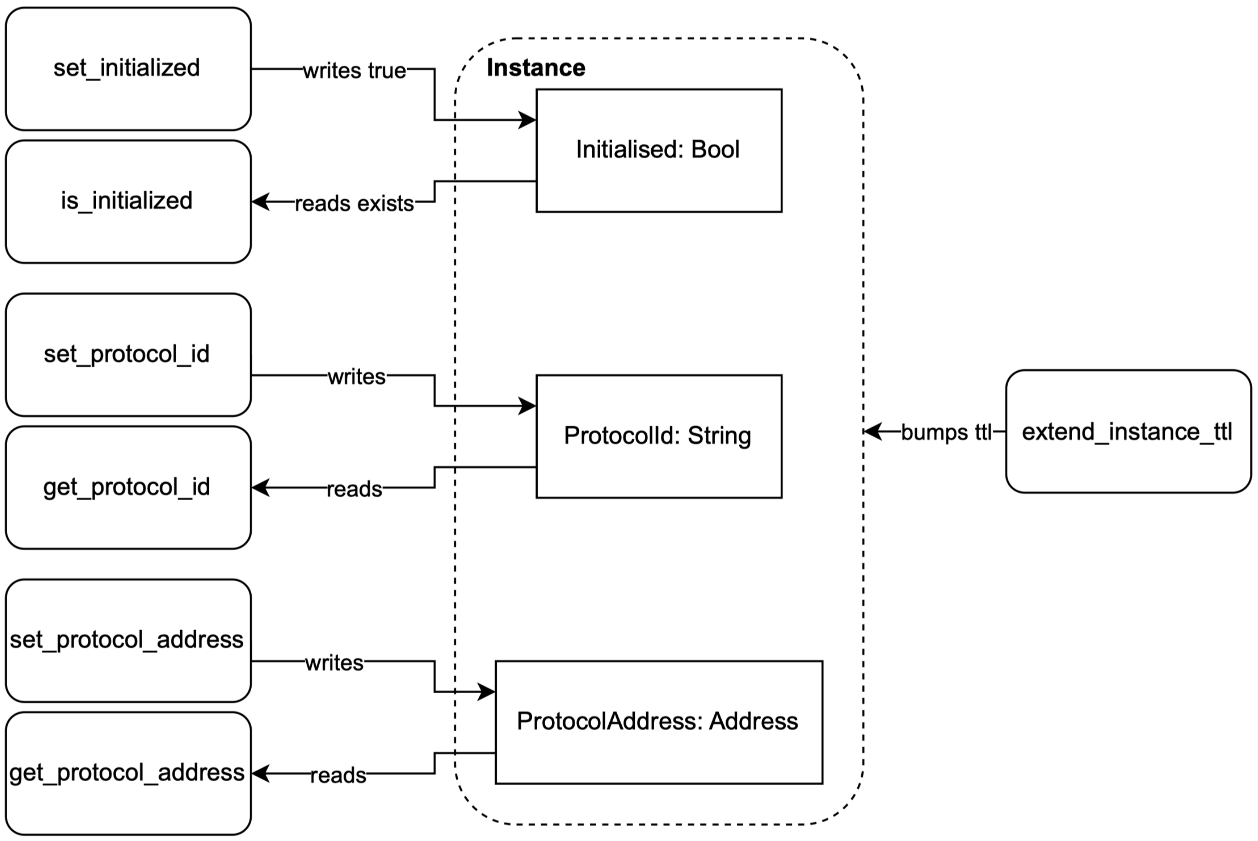

Figure 7: Adapter functions in relationship to storage usage (1).

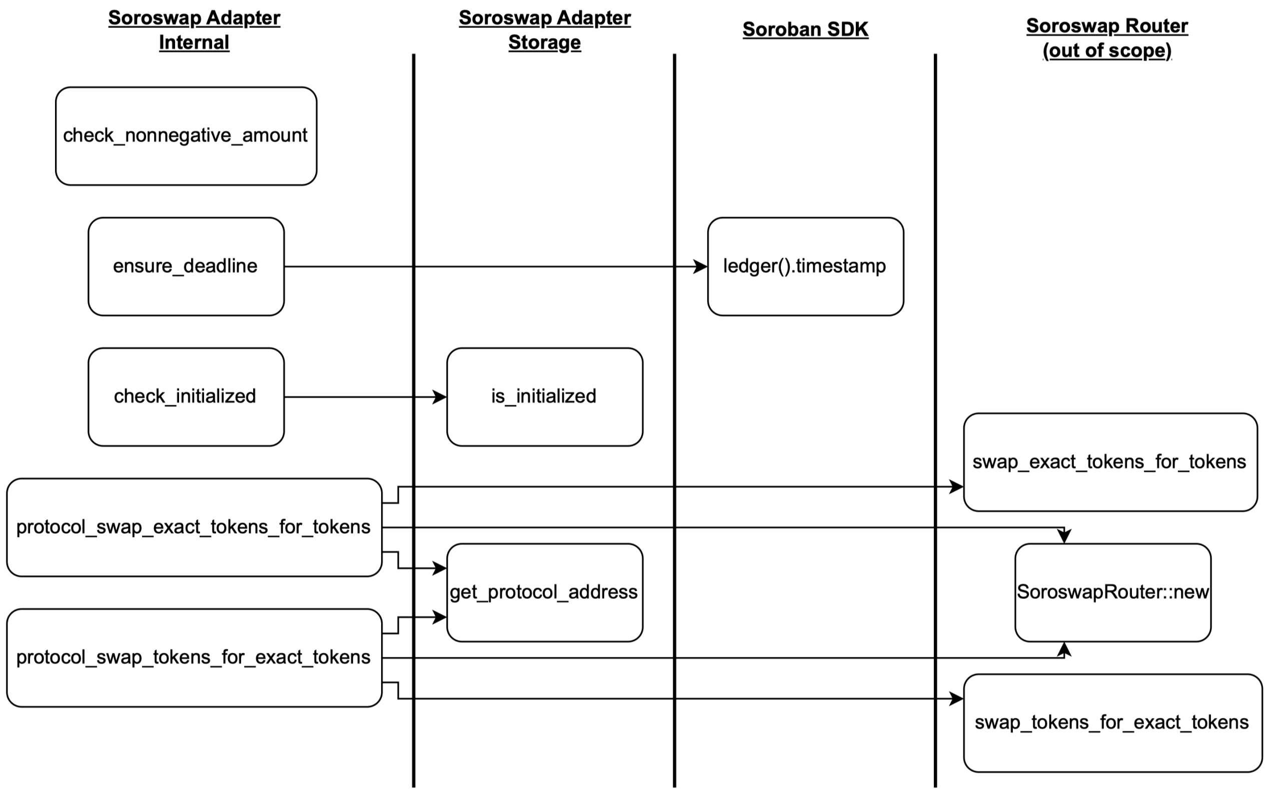
Soroswap Adapter Internal
Soroswap Adapter Storage
Soroban SDK
Soroswap Router (out of scope)
check_nonnegative_amount
ensure_deadline
ledger().timestamp
check_initialized
is_initialized
swap_exact_tokens_for_tokens
protocol_swap_exact_tokens_for_tokens
get_protocol_address
SoroswapRouter::new
protocol_swap_tokens_for_exact_tokens
swap_tokens_for_exact_tokens

Figure 8: Adapter functions in relationship to storage usage (2).

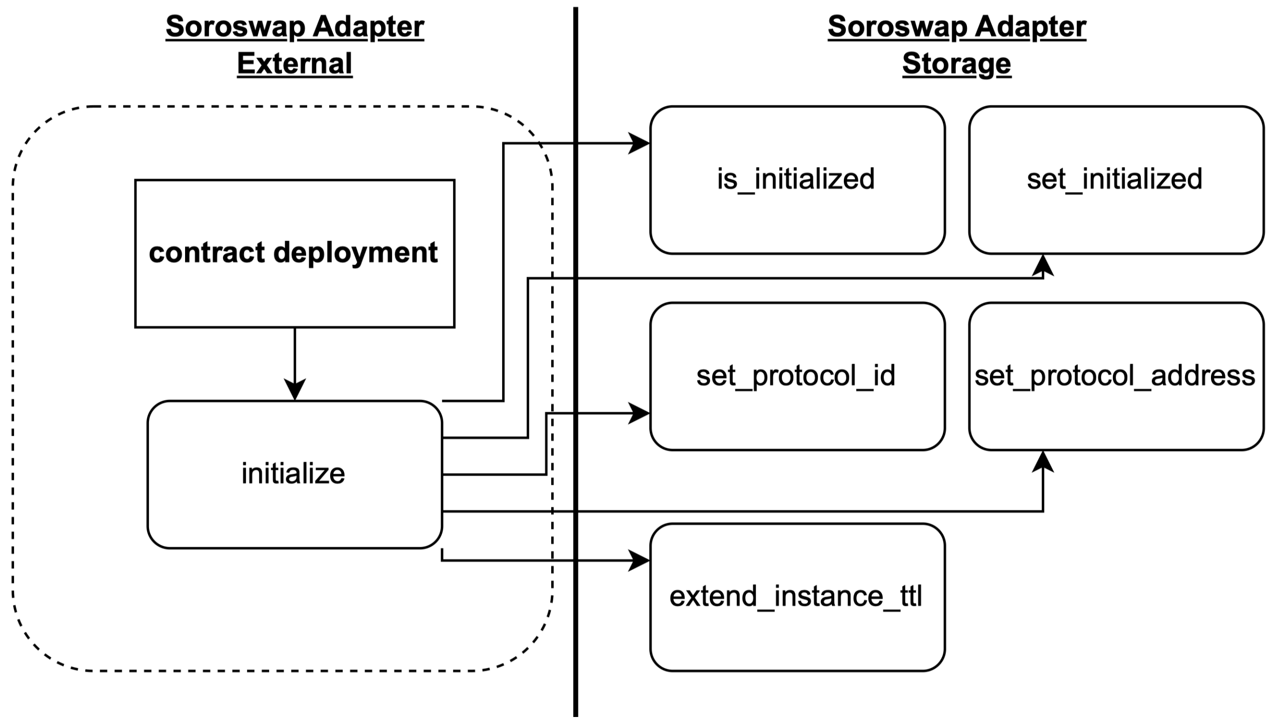

Figure 9: Adapter functions in relationship to storage usage (3).

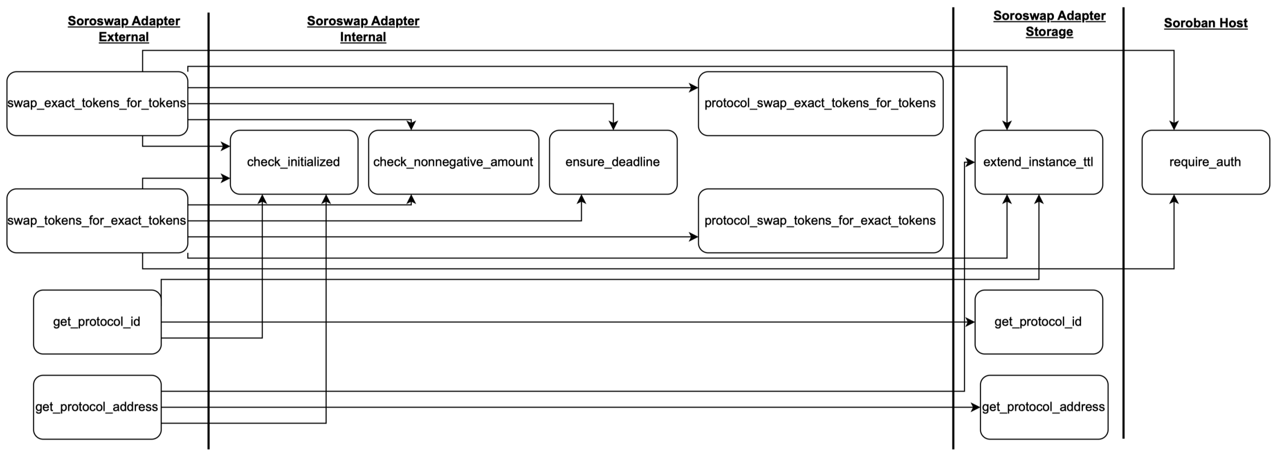

Figure 10: Adapter function and storage diagram.

Phoenix Adapter Storage

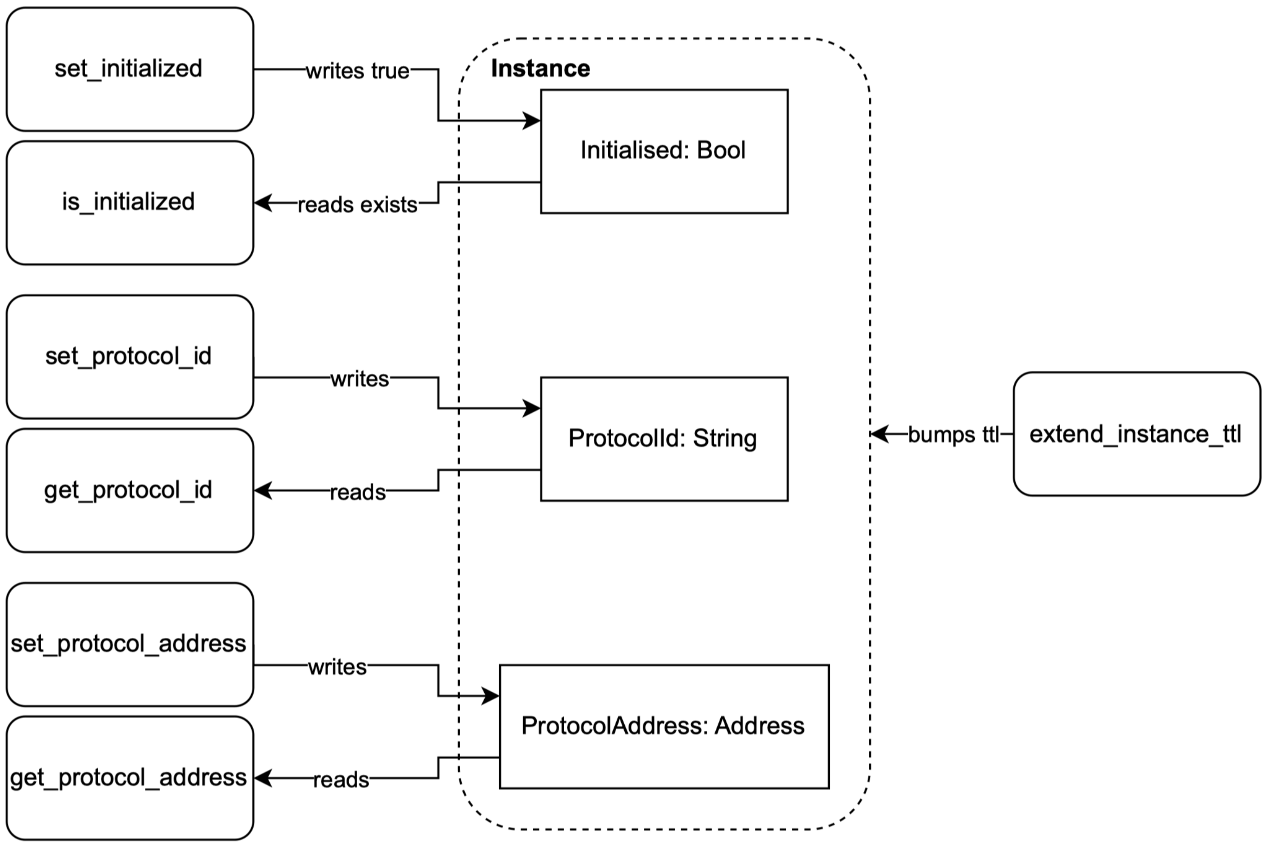

Figure 11: Phoenix adapter functions in relationship to storage (1).

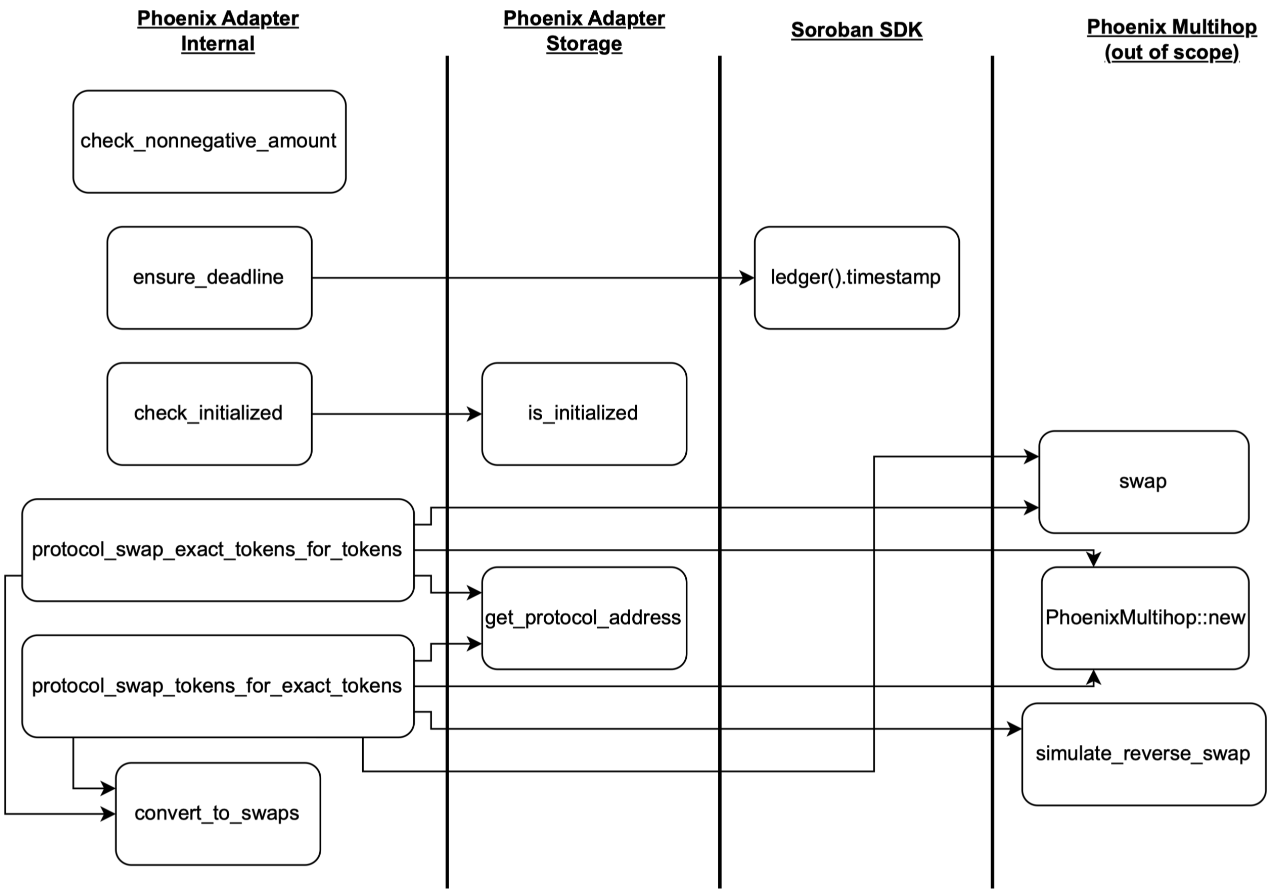

Figure 12: Phoenix adapter functions in relationship to storage (2).

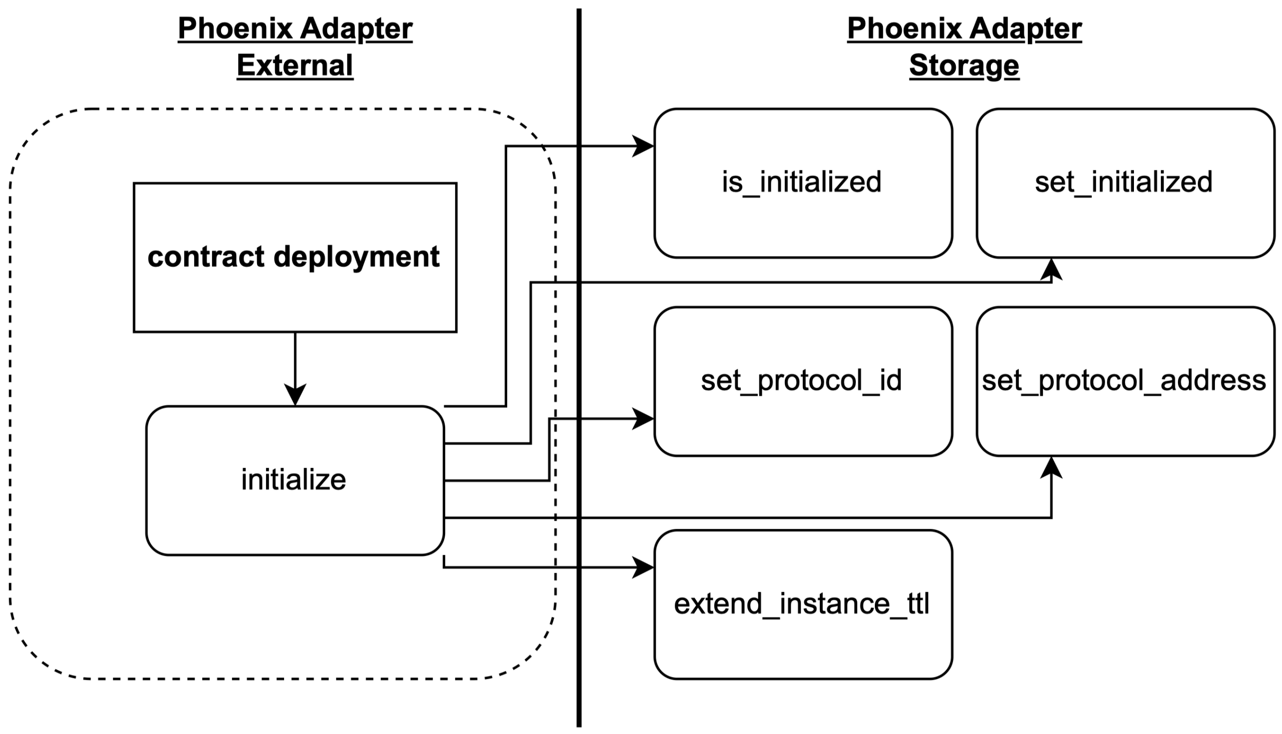

Figure 13: Phoenix adapter functions in relationship to storage (3).

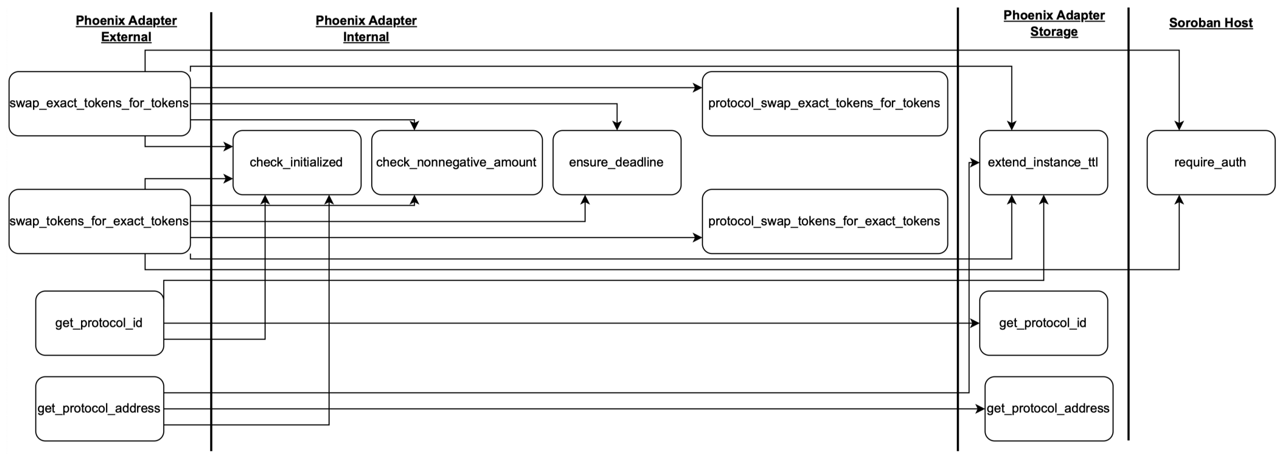

Figure 14: Adapters function diagram.