Security Audit Report

Band Protocol: Soroban - Band Standard Reference Contract Stellar

Delivered: February 19, 2024

Prepared for Band / Stellar by

Table of Contents

Executive Summary

Band Protocol engaged Runtime Verification Inc. to conduct a security audit of their Soroban - Band Standard Reference Contract crate. The objective was to review the platform's business logic and implementation in Rust and identify any issues that could cause the system to malfunction or be exploited.

The audit was conducted between 10th January 2024 and 29th January 2024. and focused on analysing the security and correctness of the source code of Band's centralised oracle system for Soroban / Stellar. The system enables users to read the Band Protocol price of supported assets, which are stored in the contract's temporary storage.

Goal

The goal of the audit is threefold:

1. Review the high-level business logic (protocol design) of Band Protocol's system based on the provided documentation;
2. Review the low-level implementation of the system for the individual Rust modules and functions within for the std-reference crate;
3. Analyse the integration between the modules and functions 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 the audit is limited to the code contained in the repository provided by the client. The single repository provided was:

- Soroban - Band Standard Reference Contract (Commit c21f943249d79608994e89f2ea03b4a5f090f080): implements the code that assigns contract `admin` and `relayers`, updates storage with `rate`s for `symbol`s, reads the prices as `RefData` or `ReferenceData` from storage.

The comments provided in the code, and 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.

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 analysed all the proposed features of the platform and the actors involved in the lifetime of a deployed contract.

Second, we thoroughly reviewed the crate source code to detect any unexpected (and possibly exploitable) behaviours. To facilitate our understanding of the platform's behaviour, higher-level representations of the Rust codebase were created, where the most comprehensive were:

- Manually built high-level function call maps and diagrams, aiding the comprehension of the code and organisation of the protocol's verification process;
- Formally specified the behaviour of each component in the system, considering the limitations of size and types of variables of the utilised modules, checking if all desired properties hold for any possible input value;
- Manually formally verified the correctness of key specifications using weakest precondition reasoning;
- Developed invariants that ought to hold for an ideal system, and reasoned their correctness against the previously referred specifications;
- Modified and created tests to search and identify possible issues in Band Protocol's logic, and to observe the correctness of specifications concretely;

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

Tools capable of identifying dependency-related issues such as cargo-audit have been used to analyse possible security issues in crates referenced in this code.

Finally, we performed rounds of internal discussion with security experts and over the code and platform design, aiming to verify possible exploitation vectors and to identify improvements for the analysed contracts.

Additionally, we discussed edge cases and particulars of the Soroban / Stellar design with SDF developers and engineers, to ensure that the behaviours of the system in environment were understood and modelled correctly.

Platform Features Description

Band Protocol's Soroban Standard Reference Contract (SSRC) acts as a centralised oracle of price data of supported assets from Band Chain. These assets are uniquely identified by symbols. After deployment, there are 3 access levels to the contract, `Admin`, `Relayer`, and `User`. The information flow security lattice for the 3 levels is $User \leq Relayer \leq Admin$. The `Admin` can grant or remove the `Relayer` privilege to addresses. `Relayer`s are able to store a symbol and it's corresponding rate in temporary storage using struct `RefData`. It is intended that these symbols and rates are feed directly from data on Band Chain, although no explicit connection exists in the SSRC. Assuming that the values are still live in storage, `Users` are able to either read the stored rates directly as `RefData`, or manually calculate a rate of two selected symbols using another struct `ReferenceData`.

All externally callable methods are contained in contract.rs. A brief description of each externally callable function, and their permissible security level is as follows:

`Admin`

- `upgrade` - upgrades the SSRC to point to new wasm bytecode
- `transfer_admin` - removes the current `Admin` and adds a new `Admin`
- `add_relayers` - adds `Relayer` privilege to provided addresses
- `remove_relayers` - removes `Relayer` privilege to provided addresses

`Relayer` and `Admin`

- `relay` - updates the `RefData` entry for provided symbols with provided rates, if the incoming data is newer than the stored data
- `force_relay` - updates the `RefData` entry for provided symbols with provided rates, regardless of the incoming data is newer than the stored data
- `delist` - removes `RefData` entries from storage for provided symbols

`User`, `Relayer`, and `Admin`

- `version` - returns the current version of the SSRC
- `address` - returns the address of the SSRC contract

- `current_admin` - returns the address of the current `Admin`
- `is_relayer` - checks if a provided address is a relayer
- `get_ref_data` - returns the `RefData` entries for provided symbols
- `get_reference_data` - returns the `ReferenceData` calculated from the `RefData` of two provided symbols `base` and `quote`

Note that `init` is not included as it is a special case function that is *intended* to be called atomically with deployment. `init` assigns `Admin` privilege to a provided address, if it has not already been assigned.

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.

Invariants

Here are listed invariants that are intended to hold of the lifetime of the system, they are separated into two categories:

- those that have been reasoned to be correct in accordance with the derived specifications
- those that cannot be satisfied with the current specifications, and thus cannot be satisfied with the current implementation

Correct:

1. After `init` is called, there is always exactly 1 `admin`.
2. After `init` is called, only the current `admin` can transfer `admin` to another address.
3. `init` can only be called 0 or 1 times.
4. Only relayers can call `relay` or `force_relay`.
5. `relay` only updates storage with new `RefData` for non-USD symbol if it has a newer `resolve_time`, or is a new symbol.
6. `force_relay` updates storage with new `RefData` for non-USD symbol independent of if it's `resolve_time`.
7. Storage never contains a `RefData` value for key `"USD"`.
8. `read_ref_data(_, "USD")` always returns a `RefData` with `rate == 10^9`, `request_id == 0`, and `resolve_time` set to the unix timestamp of last ledger entries finalisation.
9. Symbols that have been delisted can be relisted.
10. A removed `admin` is also removed as a relayer.
11. A removed `admin` can become a relayer or `admin` again.
12. A removed relayer can become a relayer again, or become an `admin`.
13. Only the admin can upgrade the contract.

Incorrect:

1. A deployed contract cannot call any other method until `init` is called.

   - Methods `version` and `address` can be called, however `version` only returns a constant that is the contract version, and `address` is addressed in Informative Finding [B1]

2. 0 < `RefData::rate`.

- Addressed in Finding [A5]

3. 0 < `ReferenceData::rate`.

- Addressed in Finding [A5]

4. `admin` is a relayer.

- Addressed in Finding [A2]

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).

Two pull requests (#15, #17) were merged to the Band SSRC repository to address the findings. We inspect the update and note whether the finding is addressed and make comment where appropriate. However, it is important to understand that this inspection is cursory, and there has not been another round of analysis performed on the updated code.

[A1] Potential denial of service (DoS) attack on `relay()` function call

Severity: High | Difficulty: Low | Recommended Action: Fix Code | Addressed by client

bandprotocol/band-std-reference-contracts-soroban/src/contract.rs

Line 148 to 154 in c21f943

```
148     fn relay(
149         env: Env,
150         from: Address,
151         symbol_rates: Vec<(Symbol, u64)>,
152         resolve_time: u64,
153         request_id: u64,
154     ) {
```

In the above code, when adding a symbol rate through the `relay()` function call, the argument `resolving_time` is an absolute time. The design logic behind this `resolving time` was to relay the rate timestamp from the Bandchain (where oracle price was provided) through a bridge. However, there is no guarateen in this contract that this `resolving time` was the real time stamp. This is to say, this parameter could suffer from the middle-man attack (for example, a malicious relayer node) and could be any value.

In the case if this parameter is set up to a far future time, e.g., like a month's equivalent time, it could block any updates to the symbol's rate before that time. This is considered a DoS attack.

Mitigation: There might many ways to mitigate this potential risk. One possible way is to set up a upper limit (i.e., a threshold) of how advanced the new resolving time can be than the exisiting timestamp. This threshold could be selected among these options: 1. how often the new oracle price is generated; 2. how often new rate is expected to be relayed; 3. how long the rate is expected to live in this contract; 4 or a combination of the previous three. It is up to the development team to decide a reasonable solution.

[A2] `new_admin` was not given the Relayer role when `transfer_admin()` was called

Severity: Low | Difficulty: Low | Recommended Action: Fix Code | Addressed by client

With regard to one of the design goals that, the admin should be granted all the subroles in the contract, in this case, the relayer role.

Thus, there should be an `add_relayer` call for the `new_admin` after the `write_admin()` call.

bandprotocol/band-std-reference-contracts-soroban/src/contract.rs

Line 105 in c21f943

```
105         write_admin(&env, &new_admin);
```

[A3] `force_relay` should have limited access to avoid unwanted overwrites

Severity: High | Difficulty: Low | Recommended Action: Fix Design | Addressed by client

The `force_relay()` function is an invocable contract function intended for emergency rate data recovery. However, as the following code implements,

bandprotocol/band-std-reference-contracts-soroban/src/contract.rs
Line 192 in c21f943

```
192         if !is_relayer(&env, &from) {
```

the current access right to `force_relay()` function is granted to all relayers, where relayers are intended to be any trusted, whiteliested addresses.

Since `force_relay()` would overwrite the rate data for a group of symbols bypassing any data validation, the current access control mechanism leaves the attack surface too large to manage. Any of the relayers if they become malicious or being compromised, could containminate the rate data or causing DoS attacks.

Mitigation: It is recommended to limit the access to highly trusted roles such as the Admin.

[A4] `relay` function will always fail if `MaxTTL` is set to be the ledger's `max_entry_ttl`

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

bandprotocol/band-std-reference-contracts-soroban/src/storage/ttl.rs

Line 10 to 11 in c21f943

```
10        let max_allowable_ttl = env.storage().max_ttl();
11       if ttl > max_allowable_ttl {
```

In the above code, the `max_allowable_ttl` = `li.max_entry_ttl`, where `li` is the ledger.

When `MaxTTL = max_allowable_ttl`, setting a new `ref_data` will always fail due to `extend_ttl` will failure.

This is because of the condition checking in the host function, `extend_ttl` will return an error when `new_live_until > max_live_until` $\land$ `durability != ContractDataDurability::Persistent`.

while in our chase,

`new_live_until = li.sequence_number + MaxTTL`

`max_live_until = li.sequence_number + li.max_entry_ttl - 1`

`durability = storage_type = ContractDataDurability::Temporal`

Mitigation:

This issue is better catogorised to a confusing implementation of the host function. We've reported an issue to the soroban development team. For the Band team, waiting for a fix from soroban team is one option Otherwise, changing the condition checking from `ttl > max_allowable_ttl` to `ttl >= max_allowable_ttl` would be able to exclude this corner case.

[A5] Missing check for non-zero `rate` in `RefData::new` and `ReferenceData::new`

Severity: Low | Difficulty: Low | Recommended Action: Fix Code | Addressed by client

Problem:
`RefData` update and unchecked_update enforce that `RefData::rate` is not zero. However, RefData::new allows `RefData::rate` to be set to zero.

The same may be true for ReferenceData::new.

Recommendation:
Since NonZeroU64 is not available as a type in Soroban, the same check must be added to `RefData::new`.

If this check is added, then RefData::unchecked_update can likely be removed, as the only function that calls `unchecked_update` is force_relay which can have lines 198 - 204 simplified to

```
RefData::new(rate, resolve_time, request_id).set(&env, symbol);
```

for the same resulting state.

[A6] Users are recommended to verify the contract logic before invoking the contract functions

Severity: Informative | Difficulty: Medium | Recommended Action: Document Prominently | Not addressed by client

The `band protocol standard reference` contract under audit is upgradable. The contract admin can upgrade the contract at any time to any logic without external authroisation. It is highly recommended for the user to verify the contract before interacting with it.

To verify the contract, the user could make use of the following checklist,

1. Examine the results of the `prepareTransaction` before signing to verify if the transaction result meets expectation;
2. Examine the contract information to see if it is updated after your last interaction. The API `getLedgerEntries` could provide the necessary information needed.
3. If the contract is upgraded after last known, please check if the new contract has been auditted by a trustworthy 3rd party.

Disclaimer: The above checklist is not meant to be exhaustive and does not guarantee the safety/security of your accounts.

EDIT Comment on addressing this finding:
Band Protocol team provided a statement, "Noted, while we plan to make sure the version is bumped on upgrade there's no way to enforce this on Soroban's side.".

[A7] Use a smaller threshold to avoid frequent write to the ttl storage

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

Source: src/storage/ttl.rs lines 22 - 23

```
let max_ttl = env.storage().max_ttl();
 env.storage().instance().extend_ttl(max_ttl, max_ttl);
```

In the above code, `max_ttl` is equivalent to `host.LedgerInfo.max_entry_ttl`.

The code above was meant to extend the lifetime of the current contract instance to the maximum allowable ledger number under the conditions that:

1. The contract instance and code is still alive at the current ledger: `old_live_until_ledger > host.LedgerInfo.sequence_number`;
2. The exisiting number of ledgers to expire is less than the threshold: `old_live_until_ledger - host.LedgerInfo.sequence_number <= threshold`, where `threshold` = `max_ttl` in this case.

In the second condition, the frequency of a successful extension goes together with the threshold, i.e., the bigger the threshold, the more frequent of ttl bumping. Since each successful bumping comes with a fee to write the storage, according to SDF developpers. Thus, the current threshold, set as the `max_ttl`, would cause more cost than needed when bumping the ttl.

Recommendation:
In order to maintain a reasonable cost of bumping ttl and keep instance storage alive as much as possible, a reasonable threshold can be a value close to the frequency of intended interacting with the contract. Here, "intended interacting" means invoking the function which bumps the ttl.
For example, if for each invocation of the contract, bump ttl is executed.
To extend the ttl to 1000 ledger later, if the threshold is set to be 1000 ledgers, there will be a write of this ttl value at each ldger, where the cost of reaching 1000 ledgers later would be `1000 * cost(w) + 1000 * rent_per_ledger`. However, if the threshold could be a smaller number like 10, the cost reaching 1000 ledgers later would be `1 * cost(w) + 1000 * rent_per_ledger`. A smaller threshold would save more writing costs.

Disclaimer: The number 10 is an assumed example rather than a recommended value. The developer should choose a value reasonable to their intended design.

EDIT Comment on addressing this finding:

The solution provided allows for customisation of the `threshold`, however this means that it is the responsibility of the `admin` to ensure that the value is appropriate for the amount of use the contract has.

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 deviates from best practices.

Two pull requests (#15, #17) were merged to the Band SSRC repository to address the informative findings. We inspect the update and note whether the informative finding is addressed and make comment where appropriate. However, it is important to understand that this inspection is cursory, and there has not been another round of analysis performed on the updated code.

[B1] The contract function `address()` to get the contract id seems redundant

Severity: Informative | Difficulty: Low | Recommended Action: Fix Code | Addressed by client

The following invocable contract function `address()` returns the contract id.
It seems redundant since the contract id must be known before contract is called.

bandprotocol/band-std-reference-contracts-soroban/src/contract.rs
Line 81 to 83 in c21f943

```
81      fn address(env: Env) -> Address {
82          env.current_contract_address()
83      }
```

Recommended to remove it.

[B2] The effect of `delist` function can be easily wiped out by `relay` or `force_relay`

Severity: Informative | Difficulty: Low | Recommended Action: Fix Design | Not addressed by client

In the contract, `delist` function is primarily used to indicate a symbol which will not be relayed anymore to the consumer. The design intention is to "allow the user distinguishing if the data is stale or will not be supported going forward".
While in this contract implementation, relist a symbol is implicitly done in a `relay` or a `force_relay` function call under the condition if this symbol is not found in storage.

This will bring up a problem that, when there is a `relay` transaction right after the `delist` function for the same symbol, the effect of the `delist` function will be void right away. Similar for the case of `force_relay`. Since any relayers can relay and delist, this scenario is likely to happen.

Recommendation:
To avoid this scenario, it is recommended to have a separate `relist` function, making relisting a symbol explicit.

EDIT Comment on addressing this finding:
The Band Protocol team have declined to address this informative finding currently, relying on the centralisation of the `admin` and `relayers` to coordinate the transactions effectively. While this may offer some mitigation to the problem, it is still entirely possible for a race to occur between `delist` and `relay`, with the `delist` being nullified if it wins the race.

[B3] `MaxTTL` is not configurable

Severity: Informative | Difficulty: Low | Recommended Action: Fix Code | Addressed by client

The only method that writes to `DataKey::MaxTTL` in instance storage is write_max_ttl, and this method is only called in init with parameter `max_ttl` as the value written. This means that once the contract is initialised, the value of `MaxTTL` can never be changed. This presents a problem if need for a more precise maximum timeout is needed, or if an erroneous one is provided when initialising.

Recommended mitigation is to add a method to update the value.

Specifications

Listed here are formal specifications for the externally callable functions of contract.rs. The specifications for key functions have been proven correct using weakest precondition reasoning, however some assumptions were made. These assumptions are:

1. The behaviour we specified for the behaviour of the underlying Soroban / Stellar environment is correct. To formally verify the correctness of the underlying system is out of scope of this audit.
2. The preconditions for internal calls for `bump_instance_ttl_to_max` are met, and the effects of the method occur. This is encapsulated by predicate `P(bump_instance_ttl_to_max)` , which has the following properties:
    - requires the contract code and contract data exist in Instance storage
    - requires the contract code and contract data has not expired
    - requires no overflow occurs in the calculation of the new expiry data
    - ensures the contract code and contract data expiry date is updated to the newly calculated expiry date
    - ensures only the contract code and contract data expiry dates are changed
3. `upgrade` behaviour is simplified
4. `env.storInst[index]` gets the entry in instance storage of `env` at `index`
5. `env.storTemp[index]` gets the entry in temporary storage of `env` at `index`
6. `ret` is the return value of the method

---

let *li* = the LedgerInfo of `env` **in**
reads *li*.max_entry
modifies `env` .storInst[DataKey::Admin].0 ∧ `env` .storInst[DataKey::MaxTTL].0 ∧ env.storInst[DataKey::Relayer(`admin_addr`)].0
requires `env` .storInst[DataKey::Admin].0.isNone() ∨ `env` .storInst[DataKey::MaxTTL].0.isNone()
requires `max_ttl` ≤ *li*.max_entry
ensures `env` .storInst[DataKey::Admin].0 = Some(`admin_addr`)
ensures `env` .storInst[DataKey::MaxTTL].0 = Some(`max_ttl`)
ensures P(bump_instance_ttl_max)

ensures env.storInst[DataKey::Relayer(`admin_addr`)].0 = Some(())
init(`env` : Env, `admin_addr` : Address, `max_ttl` : u32)

---

reads `env` .storInst[DataKey::Admin].0
requires `env` .storInst[DataKey::Admin].0.isSome()
requires msg.sender = `env` .storInst[DataKey::Admin].0.unwrap()
ensures `env` .storInst[*contract_id*] = `new_wasm_hash`
upgrade(`env` : Env, `new_wasm_hash` : BytesN<32>)

---

ensures ret = VERSION
version() -> u32

---

reads `env`
ensures ret = `env` .current_contract_address()
address(`env` : Env) -> Address

---

reads `env` .storInst[DataKey::Admin].0
requires `env` .storInst[DataKey::Admin].0.isSome()
ensures ret = `env` .storInst[DataKey::Admin].0.unwrap()
current_admin(`env` : Env) -> Address

---

let *current_admin* = old(`env`).storInst[DataKey::Admin].0.unwrap() **in**
modifies `env` .storInst[DataKey::Admin].0 ∧ `env` .storInst[DataKey::Relayer(*current_admin*)].0
requires `env` .storInst[DataKey::Admin].0.isSome()
requires msg.sender = *current_admin*
ensures `env` .storInst[DataKey::Admin].0 = Some(`new_admin`)
ensures P(bump_instance_ttl_to_max)
ensures `env` .storInst[DataKey::Relayer(*current_admin*)].0.isNone()
transfer_admin(`env` : Env, `new_admin` : Address)

---

reads `env` .storInst[DataKey::Relayer(`address`)].0 ∧ `env` .storInst[DataKey::Admin].0
requires `env` .storInst[DataKey::Admin].isSome()
ensures ret = `env` .storInst[DataKey::Relayer(`address`)].0.isSome()
is_relayer(`env` : Env, `address` : Address) -> bool

---

let *current_admin* = old(`env`).storInst[DataKey::Admin].0.unwrap() **in**
reads `addresses`
reads `env` .storInst[DataKey::Admin].0
modifies $\bigwedge_{i=0}^{addresses.len-1}$ `env` .storInst[DataKey::Relayer(`addresses` [i])]
requires `env` .storInst[DataKey::Admin].0.isSome()
requires msg.sender = *current_admin*
ensures ∀ i:uint. i < `addresses` .len $\Longrightarrow$ `env` .storInst[`addresses` [i]].0 = Some(())
ensures P(bump_instance_ttl_to_max)
add_relayers(`env` : Env, `addresses` : Vec<Address>)

---

let *current_admin* = old(`env`).storInst[DataKey::Admin].0.unwrap() **in**
reads `addresses`
reads `env` .storInst[DataKey::Admin].0
modifies $\bigwedge_{i=0}^{addresses.len-1}$ `env` .storInst[DataKey::Relayer(`addresses` [i])]
requires `env` .storInst[DataKey::Admin].0.isSome()
requires msg.sender = *current_admin*
ensures ∀ i:uint. i < `addresses` .len $\Longrightarrow$ `env` .storInst[`addresses` [i]].0.isNone()
remove_relayers(`env` : Env, `addresses` : Vec<Address>)

---

let *refData[i]* = λi:uint. `env` .storTemp[RefData::(`symbol_rates` [i].0)].0 *in*
let *refDataTTL[i]* = λi:uint. `env` .storTemp[RefData::(`symbol_rates` [i].0)].1 *in*
reads `symbol_rates` ∧ `env` .storInst[DataKey::Admin] ∧
`env` .storInst[DataKey::Relayer(from)] ∧ `env` .storInst[DataKey::MaxTTL]
modifies $\bigwedge_{i=0}^{symbol_{r}ates.len-1}$ (`symbol_rates` [i].0 $\neq$ "USD" $\Longrightarrow$

`env`.storTemp[DataKey::RefData(`symbol_rates`[i].0)])
requires `env`.storInst[DataKey::Admin].0.isSome() ∧ `env`.storInst[DataKey::Relayer(`from`)].0.isSome() ∧ `env`.storInst[DataKey::MaxTTL].0.isSome() ∧ msg.sender = `from` ∧ `rate` ≠ 0
// if the symbol is USD, then storage doesn't change
ensures ∀ i:uint. i < `symbol_rates`.len ∧ `symbol_rates`[i].0 = "USD" ⟹ *refData[i]* = old(refData[i])
// if the symbol is not USD, the symbol exists in storage already, but the resolve times overlap, storage is not updated
ensures ∀ i:uint. i < `symbol_rates`.len ∧ `symbol_rates`[i].0 ≠ "USD" ∧ old(*refData[i]*).isSome() ∧ `resolve_time` &leq: old(*refData*)[i].unwrap().resolve_time ⟹ *refData[i]* = old(*refData[i]*)
// if the symbol is not USD, the symbol is in storage already, resolve times do not overlap, there are no duplicates, update storage
ensures ∀ i:uint. i < `symbol_rates`.len ∧ `symbol_rates`[i].0 ≠ "USD" ∧ old(*refData[i]*).isSome() ∧ old(*refData[i]*).unwrap().resolve_time < `resolve_time` ∧ (∄ j:uint. j < i ∧ `symbol_rates`[i].0 = `symbol_rates`[j].0) ⟹ *refData[i]*.unwrap() = rd:RefData & *refDataTTL[i]* = `env`.storInst[DataKey::MaxTTL].0.unwrap() ∧ rd.rate = `symbol_rates`[i].1 ∧ rd.resolve_time = `resolve_time` ∧ rd.request_id = `request_id`
// if the symbol is not USD, the symbol is not in storage already, there are no duplicates, assign storage is newly assigned
ensures ∀ i:uint. i < `symbol_rates`.len ∧ `symbol_rates`[i].0 ≠ "USD" ∧ old(*refData[i]*).isNone() ∧ (∄ j:uint. j < i ∧ `symbol_rates`[i].0 = `symbol_rates`[j].0) ⟹ *refData[i]*.unwrap() = rd:RefData ∧ *refDataTTL[i]* = `env`.storInst[DataKey::MaxTTL].0.unwrap() ∧ rd.rate = `symbol_rates`[i].1 ∧ rd.resolve_time = `resolve_time` ∧ rd.request_id = `request_id` ∧ fresh(rd)
// if the symbol is not USD, the symbol is not in storage already, there are duplicates, assign storage with earliest duplicate
ensures ∀ i,j:uint. j < i < `symbol_rates`.len ∧ `symbol_rates`[i].0 ≠ "USD" ∧ old(*refData[i]*).isNone() ∧ `symbol_rates`[i].0 = `symbol_rates`[j].0 ∧ (∄ k:uint. k < j ∧ `symbol_rates`[k].0 = `symbol_rates`[j].0) ⟹ *refData[i]*.unwrap() = rd:RefData & *refDataTTL[i]* = `env`.storInst[DataKey::MaxTTL].0.unwrap() ∧ rd.rate = `symbol_rates`[j].1 ∧ rd.resolve_time = `resolve_time` ∧ rd.request_id = `request_id` ∧ fresh(rd)
// if the symbol is not USD, the symbol is in storage already, resolve times do not overlap, there

are duplicates, update storage with earliest duplicate
ensures ∀ i,j:uint. j < i < `symbol_rates`.len ∧ `symbol_rates`[i].0 ≠ "USD" ∧ old(*refData[i]*).isSome() ∧ old(*refData*)[i].unwrap().resolve_time < `resolve_time` ∧ `symbol_rates`[i].0 = `symbol_rates`[j].0 ∧ (∄ k:uint. k < j ∧ `symbol_rates`[k].0 = `symbol_rates`[j].0) ⟹ *refData[i]*.unwrap() = rd:RefData ∧ *refDataTTL[i]* = `env`.storInst[DataKey::MaxTTL].0.unwrap() ∧ rd.rate = `symbol_rates`[j].1 ∧ rd.resolve_time = `resolve_time` ∧ rd.request_id = `request_id`
ensures P(bump_instance_ttl_to_max)
relay(`env` : Env, `from` : Address, `symbol_rates` : Vec<(Symbol, u64)>, `resolve_time` : u64, `request_id` : u64)

---

let *refData[i]* = λi:uint. `env`.storTemp[RefData::(`symbol_rates`[i].0)].0 *in*
let *refDataTTL[i]* = λi:uint. `env`.storTemp[RefData::(`symbol_rates`[i].0)].1 *in*
reads `symbol_rates` ∧ `env`.storInst[DataKey::Admin] ∧ `env`.storInst[DataKey::Relayer(from)] ∧ `env`.storInst[DataKey::MaxTTL]
modifies $\bigwedge_{i=0}^{symbol_{r}ates.len-1}$ (`symbol_rates`[i].0 ≠ "USD" ⟹ `env`.storTemp[DataKey::RefData(`symbol_rates`[i].0)])
requires `env`.storInst[DataKey::Admin].0.isSome() ∧ `env`.storInst[DataKey::Relayer(`from`)].0.isSome() ∧ `env`.storInst[DataKey::MaxTTL].0.isSome() ∧ msg.sender = `from` ∧ `rate` ≠ 0
// if the symbol is USD, then storage doesn't change
ensures ∀ i:uint. i < `symbol_rates`.len ∧ `symbol_rates`[i].0 = "USD" ⟹ *refData[i]* = old(refData[i])
// if the symbol is not USD, the symbol exists in storage already, there are no duplicates following, storage is updated
ensures ∀ i:uint. i < `symbol_rates`.len ∧ `symbol_rates`[i].0 ≠ "USD" ∧ old(*refData[i]*).isSome() ∧ (∄ j:uint. i < j < `symbol_rates`.len ∧ `symbol_rates`[i].0 = `symbol_rates`[j].0) ⟹ *refData[i]*.unwrap() = rd:RefData ∧ *refDataTTL[i]* = `env`.storInst[DataKey::MaxTTL].0.unwrap() ∧ rd.rate = `symbol_rates`[i].1 ∧ rd.resolve_time = `resolve_time` ∧ rd.request_id = `request_id`
// if the symbol is not USD, the symbol is not in storage already, there are no duplicates following, storage is newly assigned

ensures ∀ i:uint. i < `symbol_rates` .len ∧ `symbol_rates` [i].0 ≠ "USD" ∧ old(*refData[i]*).isNone() ∧ (∄ j:uint. j < i ∧ `symbol_rates` [i].0 = `symbol_rates` [j].0) ⟹ *refData[i]*.unwrap() = rd:RefData ∧ *refDataTTL[i]* = `env` .storInst[DataKey::MaxTTL].0.unwrap() ∧ rd.rate = `symbol_rates` [i].1 ∧ rd.resolve_time = `resolve_time` ∧ rd.request_id = `request_id` ∧ fresh(rd)

// if the symbol is not USD, the symbol is not in storage already, there are duplicates, storage is assigned with latest duplicate

ensures ∀ i,j:uint. j < i < `symbol_rates` .len ∧ `symbol_rates` [i].0 ≠ "USD" ∧ old(*refData[i]*).isNone() ∧ `symbol_rates` [i].0 = `symbol_rates` [j].0 ∧ (∄ k:uint. k < i ∧ `symbol_rates` [k].0 = `symbol_rates` [j].0) ⟹ *refData[i]*.unwrap() = rd:RefData ∧ *refDataTTL[i]* = `env` .storInst[DataKey::MaxTTL].0.unwrap() ∧ rd.rate = `symbol_rates` [i].1 ∧ rd.resolve_time = `resolve_time` ∧ rd.request_id = `request_id` ∧ fresh(rd)

// if the symbol is not USD, the symbol is in storage already, there are duplicates, storage updates with latest duplicate

ensures ∀ i,j:uint. j < i < `symbol_rates` .len ∧ `symbol_rates` [i].0 ≠ "USD" ∧ old(*refData[i]*).isSome() ∧ `symbol_rates` [i].0 = `symbol_rates` [j].0 ∧ (∄ k:uint. k < j ∧ `symbol_rates` [k].0 = `symbol_rates` [j].0) ⟹ *refData[i]*.unwrap() = rd:RefData ∧ *refDataTTL[i]* = `env` .storInst[DataKey::MaxTTL].0.unwrap() ∧ rd.rate = `symbol_rates` [i].1 ∧ rd.resolve_time = `resolve_time` ∧ rd.request_id = `request_id`

ensures P(bump_instance_ttl_to_max)

force_relay(`env` : Env, `from` : Address, `symbol_rates` : Vec<(Symbol, u64)>, `resolve_time` : u64, `request_id` : u64)

---

reads `symbols` ∧ `env` .storInst[DataKey::Admin] ∧ `env` .storInst[DataKey::Relayer(`from`)]

modifies $\bigwedge_{i=0}^{symbols.len-1}$ `env` .storTemp[DataKey::RefData(`symbols` [i])]

requires `env` .storInst[DataKey::Admin].isSome()

requires `env` .storInst[DataKey::Relayer(`from`)].isSome()

requires msg.sender = `from`

ensures ∀ i:uint. i < `symbols` .len ⟹ `env` .storTemp[DataKey::RefData(`symbols` [i])].isNone()

delist(`env` : Env, `from` : Address, `symbols` : Vec<Symbol>)

---

let *li* = the LedgerInfo of `env` **in**

reads `symbols` ∧ `env` .storInst[DataKey::Admin] ∧ *li*.timestamp ∧ $\bigwedge_{i=0}^{symbols.len-1}$ (`symbols` [i] $\neq$ "USD" $\implies$ `env` .storTemp[DataKey::RefData(`symbols` [i])])

requires `env` .storInst[DataKey::Admin].isSome()

requires ∀ i:uint. i < `symbols` .len ∧ `symbols` [i] $\neq$ "USD" $\implies$ `env` .storTemp[DataKey::RefData(`symbols` [i])].isSome()

ensures ret $=$ Ok(rds:Vec<RefData>) ∧ fresh(rds)

ensures rds.len $=$ `symbols` .len

ensures ∀ i:uint. i < `symbols` .len ∧ `symbols` [i] $\neq$ "USD" $\implies$ rds[i] $=$ `env` .storTemp[DataKey::RefData(`symbols` [i])].unwrap()

ensures ∀ i:uint. i < `symbols` .len ∧ `symbols` [i] $=$ "USD" $\implies$ rds[i] $=$ rd ∧ fresh(rd) ∧ rd.rate $= 10^9$ ∧ rd.resolve_time $=$ *li*.timestamp ∧ rd.request_id $=$ 0

get_ref_data(`env` : Env, `symbols` : Vec<Symbol>) -> Result<Vec, StandardReferenceError>

---

let *li* = the LedgerInfo of `env` **in**

reads `symbol_pair` ∧ `env` .storInst[DataKey::Admin] ∧ *li*.timestamp ∧ $\bigwedge_{i=0}^{symbol\_pairs.len-1}$ ((`symbols_pairs` [i].0 $\neq$ "USD" $\implies$ `env` .storTemp[DataKey::RefData(`symbol_pair` [i].0)]) ∧ (`symbols_pairs` [i].1 $\neq$ "USD" $\implies$ `env` .storTemp[DataKey::RefData(`symbol_pair` [i].1)].0))

requires `env` .storInst[DataKey::Admin].isSome()

requires ∀ i:uint. i < `symbol_pair` .len ∧ `symbol_pair` [i].0 $\neq$ USD $\implies$ `env` .storTemp[DataKey::RefData(`symbols_pairs` [i].0)].0.isSome()

requires ∀ i:uint. i < `symbol_pair` .len ∧ `symbol_pair` [i].1 $\neq$ USD $\implies$ `env` .storTemp[DataKey::RefData(`symbols_pairs` [i].1)].0.isSome()

requires ∀ i:uint. i < `symbol_pair` .len ∧ `symbol_pair` [i].1 != USD $\implies$ `env` .storTemp[DataKey::RefData(`symbols_pairs` [i].1)].0.unwrap() $\neq$ 0

ensures ret == Ok(rds:Vec<ReferenceData>) ∧ fresh(rds) ∧ rds.len = `symbol_pair` .len

ensures ∀ i:uint. i < `symbol_pair` .len $\implies$ rds[i].rate = $\frac{b.\mathtt{rate} \times 10^{18}}{q.\mathtt{rate}}$ ∧ rds[i].last_updated_base = *b*.resolve_time ∧ rds[i].last_updated_quote = *q*.resolve_time

ensures ∀ i:uint. i < `symbol_pair` .len ∧ `symbol_pair` [i].0 $\neq$ "USD" $\implies$ *b* = `env` .storTemp[DataKey::RefData(`symbols` [i].0)].0.unwrap()

ensures ∀ i:uint. i < `symbol_pair` .len ∧ `symbol_pair` [i].1 $\neq$ "USD" $\implies$ *q* =

`env`.storTemp[DataKey::RefData(`symbols`[i].1)].0.unwrap()

ensures ∀ i:uint. i < `symbol_pair`.len ∧ `symbol_pair`[i].0 = "USD" $\Longrightarrow$ fresh(*b*) ∧ *b*.rate = 10^9 ∧ *b*.resolve_time = *li*.timestamp

ensures ∀ i:uint. i < `symbol_pair`.len ∧ `symbol_pair`[i].1 = "USD" $\Longrightarrow$ fresh(*q*) ∧ *q*.rate = 10^9 ∧ *q*.resolve_time = *li*.timestamp

get_reference_data(`env`: Env, `symbol_pair`: Vec<(Symbol, Symbol)>) -> Result<Vec, StandardReferenceError>