Allbridge Soroban Bridge

Stellar Audit Bank

Reference	24-01-1500-REP
Version	1.2
Date	22/02/2024

Quarkslab

Quarkslab SAS
10 boulevard Haussmann
75009 Paris
France

1. Project Information

Document history			
Version	**Date**	**Details**	**Authors**
1.0	02/02/2024	Initial version	Madigan Lebreton Elouan Wauquier
1.1	05/02/2024	Erratum	Madigan Lebreton Elouan Wauquier
1.2	22/02/2024	Fixes review	Madigan Lebreton

Quarkslab		
Contact	**Role**	**Contact Address**
Pauline Sauder	Project Manager	`psauder@quarkslab.com`
Stavia Salomon	Sales	`ssalomon@quarkslab.com`
Mathieu Robert	Director of Consulting	`mrobert@quarkslab.com`

Allbridge		
Contact	**Role**	**Contact Address**
Yuriy Savchenko	N/A	`ys@allbridge.io`

Contents

2. Executive Summary

2.1 Context

This report presents the work of the collaboration between Allbridge and Quarkslab, as defined in 24-01-1474-PRO. Quarkslab's objective was to conduct a security assessment of four (4) smart contracts for a Soroban bridge.

The audit parameter was defined by the content of the following GitHub repository: allbridge-io/allbridge-core-soroban-contracts at commit `f9f56b0f8cdd5ad4d3aa63929c4294aaf264535d`.

The fixes review was based on the content of the following GitHub repository: allbridge-io/allbridge-core-soroban-contracts at commit `7638b99b72a1e29a84abb463c62e3b4d0e06c985`.

2.2 Objectives

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

2.3 Methodology

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

2.4 Disclaimer

This report reflects the work and results obtained within the duration of the audit for the specified scope in 24-01-1474-PRO as agreed between Allbridge and Quarkslab. Tests are not guaranteed to be exhaustive and the report does not ensure that the application is bug-free.

2.5 Findings Summary

Based on the aforementioned approach, one (1) vulnerability with medium severity ranking was identified during Quarkslab's assessment, as well as one (1) lower severity issue. The report describes the attack surface and items which were assessed, as well as recommendations on how to fix the above-mentioned vulnerabilities.

ID	Name	Perimeter
MED-1	Admin can drain stablecoin liquidity	Pool (`set_bridge`)

`LOW-1`	Lack of input sanitization in admin functions	Pool (setters)
`INFO-1`	Tests reproduce the code logic	Tests
`INFO-2`	Bridge implements an insecure pattern	Bridge (`swap_and_bridge`)
`INFO-3`	Superfluous storage `DataKey::ReceivedMessage` .	Messenger
`INFO-4`	Unused variant `DataKey::Admin` .	Gas oracle
`INFO-5`	Admin fees seem to be incorrect	Pool (`initialize`)
`INFO-6`	Multiple casting from u128 to i128	Pool (`deposit`)

Severity: critical, high, medium, low, info

2.6 Recommendations

ID	Recommendations	Perimeter
`MED-1`	There are several ways to mitigate this issue. For example, a consensus mechanism would limit the amount of trust in the admin. Enforcing a grace period when updating critical parameters gives time to users and developers to react in case of compromise.	Pool (`set_bridge`)
`LOW-1`	Percentage parameters should be lower than or equal to `Pool::BP` .	Pool (setters)
`INFO-1`	Replace computations in test cases with constants whenever possible.	Tests
`INFO-2`	Consider checking that a Pool contract is associated to the input token address before calling this address.	Bridge (`swap_and_bridge`)
`INFO-3`	Remove the dead code.	Messenger
`INFO-4`	Remove the dead code and simplify the resulting `DataKey` enum (single variant).	Gas oracle
`INFO-5`	Verify that the default `admin_fee_share_bp` value configured in the `Makefile` is correct.	Pool (`initialize`)
`INFO-6`	Consider checking for overflow/underflow when casting operations are performed.	Pool (`deposit`)

Severity: critical, high, medium, low, info

2.7 Fixes

On the 2024/02/22, Quarkslab reviewed the fixes implemented following the reported vulnerabilities.

ID	Name	Fix status
MED-1	Admin can drain stablecoin liquidity	∼
LOW-1	Lack of input sanitization in admin functions	✓
INFO-1	Tests reproduce the code logic	✓
INFO-2	Bridge implements an insecure pattern	✓
INFO-3	Superfluous storage `DataKey::ReceivedMessage`.	✓
INFO-4	Unused variant `DataKey::Admin`.	✓
INFO-5	Admin fees seem to be incorrect	✗
INFO-6	Multiple casting from u128 to i128	✓

Severity: critical, high, medium, low, info

Allbridge plans to migrate the `Admin` role to a DAO smart contract as part of the 2024 timeline.
Once implemented, this DAO structure will mitigate the `MED-1` issue.

The initialized value reported in `INFO-5` hasn't been modified.
This potentially incorrect value can be fixed once deployed through the `set_admin_fee_share` function.

3. Manual Review

In Soroban, smart contracts are WebAssembly modules with a specific structure. In their `import` section, they specify the host functions they need. In their `export` section, they specify which functions can be called by users and other smart contracts.

We used the following command to get an overview of the smart contracts' attack surface:

```
$ wasm-objdump -j import --details target/wasm32-unknown-unknown/release/*.wasm
$ wasm-objdump -j export --details target/wasm32-unknown-unknown/release/*.wasm
```

A summary is available in Appendix A.

3.1 Compilation

We compiled the project using Rust 1.74 and the `wasm32-unknown-unknown` target.

INFO	`INFO-1` Tests reproduce the code logic
Perimeter	Tests
Description	
When testing a functionality, tests compute values the same way as the smart contract. If there is a bug in the smart contract, it will be reproduced in the test case and won't be caught. See for example the computation of `expected_fee` in `tests/src/messenger.rs`.	
Recommendation	
Replace computations in test cases with constants whenever possible.	

3.2 Bridge

3.2.1 Purpose

The goal is to let users send tokens to another chain's bridge, receive tokens from another chain's bridge, or balance the available tokens in case they are not bridged uniformally.

Internally, the bridge swaps to and from a virtual stablecoin (named vUSD) to transfer the tokens from one chain to another.

3.2.2 Storage

The **bridge** smart contract stores the state of current and past transfers.

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

- the address of the administrator, with the symbol `symbol_short!("Admin");`

- the address of the stop authority, with the symbol `symbol_short!("StopAuth")`;
- the address of the gas oracle, with the symbol `symbol_short!("GasOrclAd")`;
- the address of the native token, with the symbol `symbol_short!("NatvTknAd")`;
- and its config as `Bridge`, with the symbol `symbol_short!("Config")`.

The configuration contains the address of the messenger smart contract, as well as the address of a designated rebalancer (which is exempt from fees). It also contains the addresses of vUSD pools for each supported token, as well as conversion factors handling the different `decimals` of each token. Finally, it has a field named `can_swap` that can pause the smart contract's operations in case of an emergency.

> Some Instance-level fields are frozen and cannot be modified after initialization. They include the admin address and the *native token*.

At the *Persistent* level, it stores the address of bridges on other chains, as well as the tokens they support in `DataKey::OtherBridge(chain_id)`. It also stores the hash of the messages it sent and processed.

> *Persistent* and *Instance* level storage entries cannot appear in the transaction's footprint if they are expired. This ensures expired entries cannot be exploited.

3.2.3 Permissioned functionality

The `initialize` method can be called only when the "Config" field in Instance storage is not set. Since the TTL of the contract instance and all globals are tied together, there is no risk of this field expiring before the contract instance.

It configures the main smart contracts the bridge interacts with (gas oracle, messenger, native token) and sets the admin's address. The pools are configured separately by the admin.

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

Most permissioned methods are straightforward, checking that either the configured admin or stop authority (when appropriate) authorized the transaction before modifying the corresponding configuration.

`add_bridge_token` and `remove_bridge_token` let the admin configure which tokens are supported on other chains' bridge.

`register_bridge` lets the admin register or modify the address of other chains' bridge.

`add_pool` lets the admin register a vUSD pool for a given token and computes the relevant conversion factors for the gas oracle and the bridge based on the token's `decimals`.

`start_swap` and `stop_swap` let the stop authority pause or resume the smart contract.

The stop authority is set by the admin in `set_stop_authority`, as well as the gas oracle, gas usage, messenger and rebalancer in their respective methods.

Finally, the admin can withdraw any token to an address it controls using `withdraw_bridging_fee_in_tokens`, and the native token using `withdraw_gas_tokens`.

3.2.4 Permissionless functionality

Users can perform three main actions on the bridge, gated by the `can_swap` flag:

- sending tokens,
- receiving tokens, and
- balancing the pools.

To balance pools, they call the `swap` method. The bridge then atomically calls the two relevant pools to first swap the sent token to vUSD, then to the received token. The authorized tokens and the pools are all configured and whitelisted by the admin, which is considered trusted.

When bridging tokens, this operation is split across both chains. The user sends tokens using the `swap_and_bridge` method, waits for the bridge to sign their message, then receives them when the `receive_tokens` method is called on the other chain.

The `swap_and_bridge` method performs the first half of the operation, swapping the sent token minus a fee to vUSD using the configured pool. The fee is transferred to the **bridge** while the **pool** handles the vUSD swap. Once the swap is complete, the bridge builds a message by hashing its components, and stores it as sent to avoid sending it multiple times. It then sends it to the **messenger**.

On the reception side, the `receive_tokens` method performs the second half of the operation. It rebuilds the message hash from its constituents and checks that the **messenger** on its chain received the signed message and did not already process it. It then performs a swap from vUSD to the target token using the configured pool and reimburses the extra gas to the recipient.

3.2.5 Authorization framework attack vector

The current implementation of the bridge uses an insecure pattern when `swap_and_bridge` is called. Indeed, the token contract passed as an input argument is called (in `convert_bridging_fee_in_tokens_to_` before checking if a pool is associated to this token (in `send_and_swap_to_v_usd`).

Due to the Soroban Authorization Framework and the call to an unverified external contract, an attack vector is available.

> We were not able to build a successful exploit of the attack vector described in the following section. However, we decided to describe this attack vector which is specific to the Soroban environment. Moreover, as the current implementation of the bridge allows this attack vector, the bridge may be vulnerable to an exploit if misconfigurations or upgrades are made.

The Soroban Authorization framework is designed in such a way that a user or a contract authorizes the call it makes and all the associated subcalls. Figure 3.1 explains this behavior.

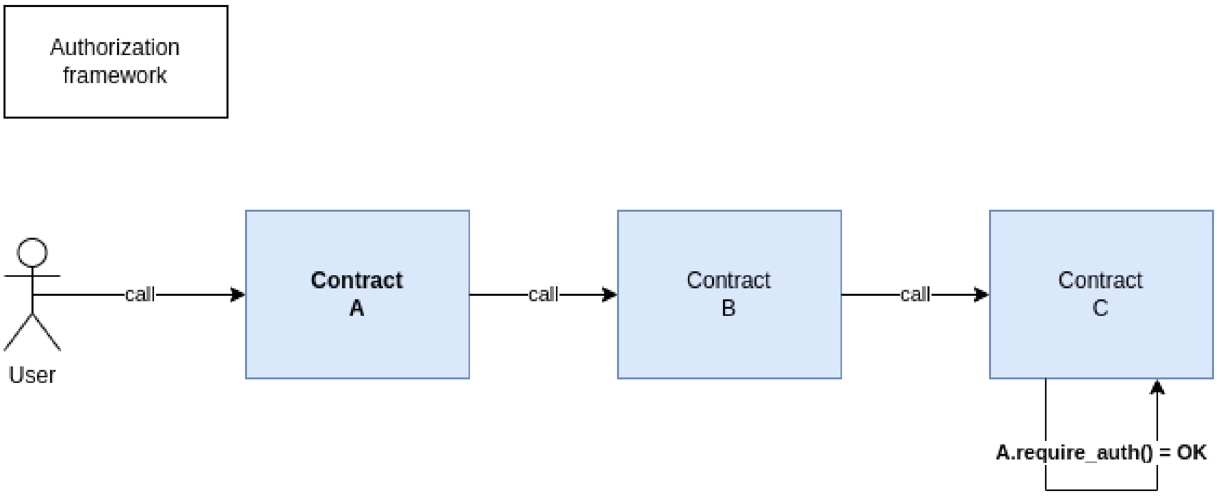

Figure 3.1: Authorization framework behavior

The bridge's current implementation makes users pass the token contract as an argument of the `swap_and_bridge` function. If this is a legit call, the interaction with the Pool contract illustrated in Figure 3.2 will occur.

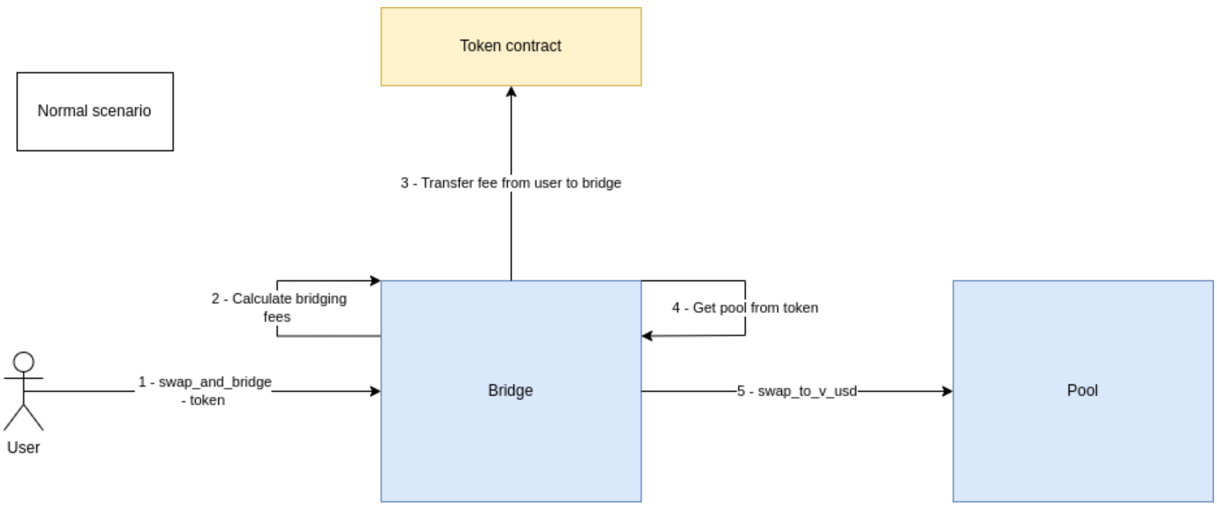

Figure 3.2: Legit swap

An attacker may pass a malicious contract as the token contract argument. As explained at the beginning of the section, this contract is called before being checked. The following interaction may occur.

Fortunately, the transaction reverts when the Pool contract address is retrieved from the user-controller token contract address.

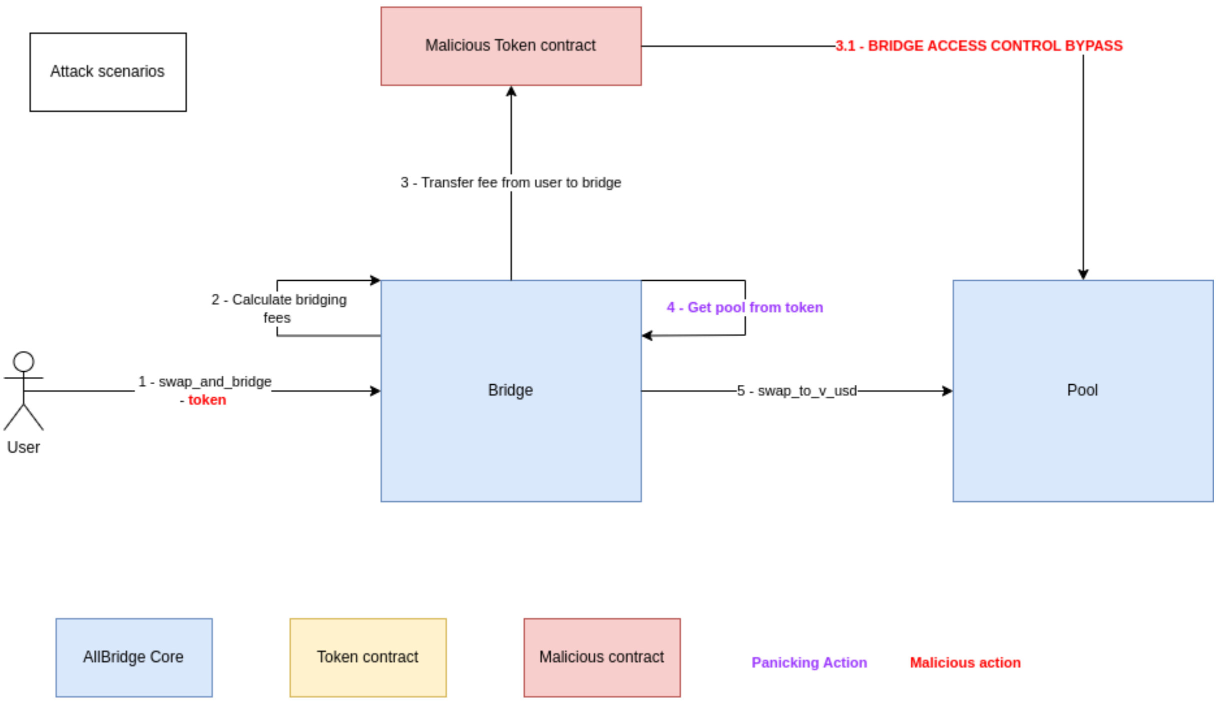

Figure 3.3: Pool access control bypass

INFO	`INFO-2` Bridge implements an insecure pattern
Perimeter	Bridge (`swap_and_bridge`)
Description	
The current implementation of `swap_and_bridge` checks that a Pool contract is associated to the user-controlled token address argument after calling this address.	
Recommendation	
Consider checking that a Pool contract is associated to the input token address before calling this address.	

A patch addressing the issue by checking the associated pool address before any other operation. It is available in Appendix B.1.

3.3 Messenger

3.3.1 Purpose

The goal is to check that messages are valid and signed by the validators, and store them for the **bridge** to process.

3.3.2 Storage

The **messenger** smart contract stores the message hashes it sends and receives.

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

- the address of the administrator, with the symbol `symbol_short!("Admin")`;
- the address of the gas oracle, with the symbol `symbol_short!("GasOrclAd")`;
- the address of the native token, with the symbol `symbol_short!("NatvTknAd")`;
- the gas usage for each supported chain, with the symbol `symbol_short!("GasUsage")`;
- and its `Config`, with the symbol `symbol_short!("Config")`.

The `Config` contains the current chain's id (should be 7), a list of supported chain ids, the public key of the primary validator, and those of the secondary validators.

> Some Instance-level fields are frozen and cannot be modified after initialization. They include the *current chain's id* and the *native token*.

> The gas usage for each supported chain is stored in a `Map<u32, u128>`. Since only 32 chains are supported, a simple array may be preferable.

> The secondary validators are stored in a `Map<BytesN<65>, bool>`, enabling the smart contract to check whether a validator is in the set in constant time. However, the `bool` value is never read and can be replaced with the unit type `()`, since only membership is checked (i.e. this is a HashSet and not a HashMap).

> The list of supported chains is stored as a bitfield, with values ranging from 0 to 31, but stored in a `BytesN<32>`, with one byte per bit. Consider using a `u32` instead for cheaper storage.

All the methods modifying Instance-level storage require the contract to be initialized (i.e. the Admin field must be set) and the authorization from the configured admin. They simply perform their advertized functionality. They are accompanied by a few getters with no special functionality either, except `get_gas_usage` which returns 0 when the field is not set.

3.3.3 Permissioned functionality

The `initialize` method can be called only when the "Config" field in Instance storage is not set. Since the TTL of the contract instance and all globals are tied together, there is no risk of this field expiring before the contract instance.

This method subsequently set all the Instance-level storage of the smart contract (see 3.3.2).

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

All the functions in `contracts/messenger/src/methods/admin` require the contract to be initialized, and the authorization from the configured admin. They are for the most part simple setters, except `withdraw_gas_tokens`. They include:

- `set_admin`,
- `set_gas_oracle`,
- `set_gas_usage`,
- `set_other_chain_ids`,
- `add_secondary_validator`,
- `remove_secondary_validator`, and
- `set_primary_validator`.

`withdraw_gas_tokens` transfers an arbitrary amount of native tokens to the provided address. These tokens are received in `send_message` and are the cumulative estimated cost of the transactions to bridge. As long as the estimated transaction costs remain fair, this method does indeed allow an admin to withdraw the funds required to bridge the transactions. A malicious admin could raise the estimated transaction cost using the **gas oracle**, collecting more funds than necessary.

3.3.4 Permissionless functionality

`send_message` lets users request the bridge to perform a transaction on another chain.

The message must be 32 bytes long, with the first two bytes specifying the origin and destination chain by id. The messenger verifies these values against `Config::chain_id` (frozen) and `Config::other_chain_ids`. Only chain ids up to and including 31 are supported.

The messenger verifies that the sender did not already send the same message to prevent replay attacks. This is achieved by storing the current ledger sequence in *Persistent* storage at a key determined by the hash of the `(message, sender)` pair.

> Persistent storage expires after some time, enabling a user to send the same message multiple times.

`receive_message` lets users request the bridge to execute a transaction coming from another chain. The message needs to be signed by the primary validator and any one of the secondary validators.

The message is then saved to *Persistent* storage.

The `Message` structure stores unnecessary value: `true` in `DataKey::ReceivedMessage`. This can be verified by removing the unnecessary details from the code.

```diff
--- a/contracts/messenger/src/storage/message.rs
+++ b/contracts/messenger/src/storage/message.rs
@@ -37,7 +37,7 @@ impl Message {
     #[allow(dead_code)]
     pub fn has_received_message(env: &Env, message: BytesN<32>) -> bool {
         let key = DataKey::ReceivedMessage(message);
-        let result = env.storage().persistent().get::<_, bool>(&key).is_some();
+        let result = env.storage().persistent().get::<_, ()>(&key).is_some();
         if result {
             Self::extend_ttl(env, &key);
         }
@@ -46,7 +46,7 @@ impl Message {

     pub fn set_received_message(env: &Env, message: BytesN<32>) {
         let key = DataKey::ReceivedMessage(message);
-        env.storage().persistent().set(&key, &true);
+        env.storage().persistent().set(&key, &());
         Self::extend_ttl(env, &key);
     }
```

INFO	`INFO-3` Superfluous storage `DataKey::ReceivedMessage`.
Perimeter	Messenger

Description

CWE-561: Dead Code

The boolean stored at `DataKey::ReceivedMessage` in `contracts/messenger/src/storage/message.rs` is never read.
Instead, the smart contract only checks whether the key exists or not.

Recommendation

Remove the dead code.

3.4 Gas oracle

3.4.1 Purpose

The goal is to provide the price of gas on the supported blockchains to the other smart contracts.

The gas price is provided by a trusted account, and anyone can read this price.

3.4.2 Storage

The **gas oracle** smart contract stores two types of information: the administrator address for authorization purposes, and the prices of each supported chain.

At the *Instance* level, the gas oracle only stores the administrator's address with the symbol key `symbol_short!("Admin")`.

It also stores the gas price and the token price for each supported chain, with the data key

`DataKey::ChainData(u32)` and a *Temporary* lifetime. This means that access to expired entries will result in an error, which is desirable for prices that can quickly become out-dated.

> The chain id is a `u32`. This means the smart contract can technically store data for chain id up to $2^{32} = 4\,294\,967\,296$, exceeding the range used by messages (i.e. a single byte).

The `DataKey` data structure contains a never-used variant: `DataKey::Admin`. This can be verified by removing the variant from the structure declaration.

```
--- a/contracts/gas_oracle/src/data_key.rs
+++ b/contracts/gas_oracle/src/data_key.rs
@@ -6,5 +6,4 @@
 #[contracttype]
 pub enum DataKey {
     ChainData(u32),
-    Admin,
 }
```

INFO	`INFO-4` Unused variant `DataKey::Admin`.
Perimeter	Gas oracle
Description	
CWE-561: Dead Code The `DataKey::Admin` variant in `contracts/gas_oracle/src/data_key.rs` is never used. Instead, the smart contract uses the `symbol_short!("Admin")` key defined in `common/bridge_storage/src/admin.rs`.	
Recommendation	
Remove the dead code and simplify the resulting `DataKey` enum (single variant).	

3.4.3 Permissioned functionality

The `initialize` method can be called only when the "Admin" field in Instance storage is not set. Since the TTL of the contract instance and all globals are tied together, there is no risk of this field expiring before the contract instance.

This method simply sets the address of the admin.

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

The `set_admin` and `set_price` methods require the contract to be initialized, and the authorization from the configured admin. They are simple setters.

3.4.4 Permissionless functionality

Anyone can read the contract storage using `get_admin` and `get_gas_price`.

The remaining function perform simple operations on these base values to provide more useful information. Overflows are not checked explicitely in code, but are enabled in `Cargo.toml`.

> Overflows result in `panic!` in `get_gas_cost_in_native_token` and `get_transaction_gas_cost_in_usd`.

3.5 Pool

3.5.1 Purpose

The Pool contract provides liquidity to the bridge contract. It allows swapping a supported stablecoin with a vUSD value in both ways.

vUSD represents a USD value inside the Allbridge Core protocol. This unit does not represent an on-chain token and is only used between the various bridges deployed by Allbridge.

3.5.2 Storage analysis

The Pool contract uses storage for multiple purposes.

The following table shows the `CONTRACT_DATA` ledger entries and their associated storage type.

Entry name	Storage type	Value stored
`Admin`	`Instance`	Address of the administrator
`Bridge`	`Instance`	Address of the bridge contract
`Pool`	`Instance`	Pool's data and settings (reserves, fees, …)
`ClaimableBalance`	`Persistent`	Balance of claimable stablecoin associated to an address
`UserDeposit`	`Persistent`	Liquidity provider's shares and paid rewards associated to an address

The storage configuration is well-defined and follows the best practices detailed in the official Soroban documentation.

3.5.3 Contract initialization

The `pool-initialize` in the `Makefile` shows the following parameters values:

Initialize parameter	Value	Interpretation
`admin`	Admin address	The administrator address of the Pool contract
`bridge`	Bridge contract address	The bridge contract allowed to perform swap in the Pool
`a`	20	Amplification coefficient used in Pool calculations
`token`	Token address	The stablecoin contract supported by the Pool
`fee_share_bp`	10	Swap fee represents 0.1% of swapped amounts
`balance_ratio_min_bp`	0	The minimum ratio between token balance and vUSD balance
`admin_fee_share_bp`	10	Administrator fee represents 0.1% of swap fees

The current `admin_fee_share_bp` value indicates that 0.1% of swap fees are allocated to the administrator. This percentage does not seem to be consistent with the configuration of existing Allbridge pools. For example, on Ethereum the administrator fee represents 20% of swap fees.

INFO	`INFO-5` Admin fees seem to be incorrect
Perimeter	Pool (`initialize`)
Description	
The current configuration indicates that 0.1% of the total collected fees are allocated to the administrator. This configuration is likely incorrect.	
Recommendation	
Verify that the default `admin_fee_share_bp` value configured in the `Makefile` is correct.	

3.5.4 Liquidity providing mechanism

The Pool contract requires a substantial stablecoin liquidity to guarantee successful funds bridging. To collect this liquidity, users are encouraged to deposit stablecoins in return for rewards.

Depositing stablecoins

Users can deposit stablecoins using the `deposit` function to increase their deposit amount.

This function transfers an amount of stablecoin from the sender to the Pool contract, calculates the increased balance in both stablecoin and vUSD liquidity.

It then updates the `d` variable which represents total liquidity in the Pool and calculates the liquidity difference between the new `d` and the old `d` liquidity.

Finally, this difference in liquidity is added to the `UserDeposit` structure associated to the sender address.

INFO	`INFO-6` Multiple casting from u128 to i128
Perimeter	Pool (`deposit`)
Description	
Multiple `u128` variables are cast to `i128` using the `as` keyword. This casting may silently overflow.	
Recommendation	
Consider checking for overflow/underflow when casting operations are performed.	

Claiming rewards

The `claim_rewards` function allows liquidity providers (users who have deposited liquidity into the pool) to collect their earned rewards.

As part of the fee, the total amount of rewards allocated to liquidity providers increases during swap operations.

When a user claims their rewards, a computation is performed to determine the reward this user is due. This computation involves multiplying the amount of liquidity shares the user holds (i.e. LPamount) by the reward amount per share. From the result of this multiplication, the amount of rewards already paid to the user is then deducted to determine the net amount to be sent to the user.

Finally, the determined net amount is paid to the user and the amount of rewards already paid is updated.

Withdrawing stablecoins

Liquidity providers can withdraw the liquidity they previously deposited by calling the `withdraw` function.

This function reduces the amount of liquidity shares held by the sender. It achieves this by modifying the `UserDeposit` structure linked to the user address and updating the total liquidity `d`. An amount of stablecoins is calculated from the liquidity amount to withdraw and the sender's accumulated rewards.

Finally, the determined amount of stablecoin is transferred from the pool to the sender address.

3.5.5 Swapping mechanism

The swapping mechanism allows the conversion between vUSD and the selected stablecoin, and conversely.

vUSD serves as a core internal unit within the Allbridge Core protocol. It acts as a medium to represent transferred funds across the multiple bridges operating on supported networks. It's crucial that external users are restricted from altering vUSD amounts.

Within the Allbridge Core protocol, only the bridge can swap stablecoin and vUSD. The Pool's swap functions are designed with access controls to ensure that the only authorized caller is the bridge.

Swapping from stablecoin to vUSD

The pool contract implements the `swap_to_v_usd` function. This function swaps an amount of stablecoin tokens to a vUSD value. This is mainly used when funds are bridged from the Soroban environment to other networks.

Only the bridge contract is allowed to call the `swap_to_v_usd` function. This access control ensures that users can't directly interact with the swap mechanism. This function correctly implements access control to exclusively authorize the address stored in the `Bridge` storage.

In most cases, the stablecoin's decimals (i.e. default is 7 in Soroban) will be greater than the vUSD decimals (i.e. 3). This may lead to users losing a negligible portion of stablecoin during each swap.

Swapping from vUSD to stablecoin

The pool contract implements the `swap_from_v_usd` function. It allows the bridge to swap vUSD to stablecoin tokens. This is mainly used when funds are bridged from other networks to the Soroban environment.

Only the bridge contract is allowed to call the `swap_from_v_usd` function. This access control ensures that users can't directly interact with the swap mechanism. This function correctly implements access control to exclusively authorize the address stored in the `Bridge` storage.

`withdraw` includes two ways to distribute the bridged funds using a `claimable` boolean input. When set to false, bridged funds are directly transferred to the receiver address. When set to true, the bridged funds amount is stored in a `ClaimableBalance` structure linked to the receiver address. Then, the `claim_balance` function allows to transfer the amount from the pool to the receiver address.

3.5.6 Administration functionalities

Several functions in the contract are defined to allow the administrator of the contract to modify configuration variables.

Function name	Action	Access control
`set_fee_share`	Set the fee percentage taken on each swap	`Admin`
`adjust_total_lp_amount`	Updates internal variables with current balances	`Admin`
`set_balance_ratio_min_bp`	Set the minimum balance ratio between stablecoin and vUSD	`Admin`
`stop_deposit`	Deactivate liquidity deposits	`StopAuthority`
`start_deposit`	Activate liquidity deposits	`StopAuthority`
`stop_withdraw`	Deactivate liquidity withdrawals	`StopAuthority`
`start_withdraw`	Activate liquidity withdrawals	`StopAuthority`
`set_stop_authority`	Set the `StopAuthority` address	`Admin`
`set_bridge`	Set the `Bridge` address	`Admin`
`set_admin`	Transfer administration permissions to a new address	`Admin`
`set_admin_fee_share`	Set the percentage of fees allocated to the `Admin`	`Admin`
`claim_admin_fee`	Claim the collected administrator fee	`Admin`

The administrator is able to modify the bridge address. The bridge address is allowed to perform swaps from vUSD values to a stablecoin amount. The administrator is able to drain part of the Pool liquidity by setting his own address as the bridge address and executing swaps using the `swap_from_v_usd` function.

MEDIUM	`MED-1` Admin can drain stablecoin liquidity		
Likelihood	●○○○	**Impact**	●●●○
Perimeter	Pool (`set_bridge`)		
Prerequisites	`Admin` role		

Description

The administrator can drain stablecoins deposited in the Pool by modifying the `Bridge` address and executing a swap using `swap_from_v_usd` .

Recommendation

There are several ways to mitigate this issue.

For example, a consensus mechanism would limit the amount of trust in the admin. Enforcing a grace period when updating critical parameters gives time to users and developers to react in case of compromise.

`set_fee_share`, `set_admin_fee_share` and `set_balance_ratio_min_bp` enable the administrator to modify percentage-type settings of the pool. Each function is lacking input sanitization to ensure that the new setting is a valid percentage.

LOW	`LOW-1` Lack of input sanitization in admin functions		
Likelihood	●○○○	**Impact**	●○○○
Perimeter	Pool (setters)		
Prerequisites			
Description			
Multiple variables set by the administrator are percentage. But the setter functions lack input sanitization to ensure that the values are not greater than 100%. Affected functions include `set_fee_share`, `set_admin_fee_share` and `set_balance_ratio_min_bp`.			
Recommendation			
Percentage parameters should be lower than or equal to `Pool::BP`.			

A patch addresses the issue by incorporating a require statement into the impacted functions. It is available in the Appendix section.

3.5.7 View functionalities

The contract implements multiple view functions.

Function name	Action
`pending_reward`	Returns the pening rewards associated to the `user` address parameter
`get_pool`	Returns the `Pool` storage structure
`get_admin`	Returns the `Admin` address
`get_stop_authority`	Returns the `StopAuthority` address
`get_bridge`	Returns the `Bridge` address
`get_user_deposit`	Returns the `UserDepost` structure associated to the `user` address parameter
`get_claimable_balance`	Returns the `u128` clamable amount associated to the `user` address parameter

These functions are publicly callable and do not implement any access control mechanism.

A. Smart contract interface

A.1 Host functions (`import` section)

In the `import` section, the host functions appear as 2 characters separated by a dot. This is done to optimize the binary size. The first character is the host module name. The second character is the name of the function inside this module. The correspondence between the compressed 1-character names and the full names can be found in Soroban's `env.json` file.

For readability reason, we translated these names in the table below.

Function name	Bridge	Messenger	Gas oracle	Pool
`address.authorize_as_curr_contract`	✓			
`address.require_auth`	✓	✓	✓	✓
`buf.bytes_copy_from_linear_memory`	✓			
`buf.bytes_copy_to_linear_memory`	✓	✓		
`buf.bytes_get`	✓	✓		
`buf.bytes_len`	✓	✓		
`buf.bytes_new`	✓			
`buf.bytes_new_from_linear_memory`	✓	✓		
`buf.bytes_put`	✓	✓		
`buf.serialize_to_bytes`	✓	✓		
`buf.symbol_new_from_linear_memory`	✓	✓	✓	✓
`call.call`	✓	✓		✓
`context.contract_event`	✓	✓		✓
`context.get_current_contract_address`	✓	✓		✓
`context.get_ledger_sequence`		✓		
`context.obj_cmp`	✓	✓		
`crypto.compute_hash_keccak256`	✓	✓		
`crypto.recover_key_ecdsa_secp256k1`		✓		

Function name	Bridge	Messenger	Gas oracle	Pool
int.obj_from_i128_pieces	✓	✓		✓
int.obj_from_u128_pieces	✓	✓	✓	✓
int.obj_to_i128_hi64	✓			
int.obj_to_i128_lo64	✓			
int.obj_to_u128_hi64	✓	✓	✓	✓
int.obj_to_u128_lo64	✓	✓	✓	✓
int.u256_val_to_be_bytes	✓			
ledger.extend_contract_data_ttl	✓	✓	✓	✓
ledger.extend_current_contract_instance_and_code_ttl	✓	✓	✓	✓
ledger.get_contract_data	✓	✓	✓	✓
ledger.has_contract_data	✓	✓	✓	✓
ledger.put_contract_data	✓	✓	✓	✓
map.map_del		✓		
map.map_get	✓	✓		
map.map_has	✓	✓		
map.map_new	✓	✓		
map.map_new_from_linear_memory	✓	✓	✓	✓
map.map_put	✓	✓		
Total (40)	**37**	**33**	**14**	**19**

A.2 Exposed functions (`export` section)

Function name	Bridge	Messenger	Gas oracle	Pool
_	✓	✓	✓	✓
add_bridge_token	✓			
add_pool	✓			
add_secondary_validator		✓		
adjust_total_lp_amount				✓
claim_admin_fee				✓
claim_balance				✓
claim_rewards				✓
crossrate			✓	
deposit				✓
get_admin	✓	✓	✓	✓
get_another_bridge	✓			
get_bridge				✓
get_claimable_balance				✓
get_config	✓	✓		
get_gas_cost_in_native_token			✓	
get_gas_oracle	✓	✓		
get_gas_price			✓	
get_gas_usage	✓	✓		
get_pool				✓
get_pool_address	✓			

Function name	Bridge	Messenger	Gas oracle	Pool
`get_price`			✓	
`get_sent_message_sequence`		✓		
`get_stop_authority`	✓			✓
`get_transaction_cost`	✓	✓		
`get_transaction_gas_cost_in_usd`			✓	
`get_user_deposit`				✓
`has_processed_message`	✓			
`has_received_message`	✓	✓		
`has_sent_message`		✓		
`initialize`	✓	✓	✓	✓
`pending_reward`				✓
`receive_message`		✓		
`receive_tokens`	✓			
`register_bridge`	✓			
`remove_bridge_token`	✓			
`remove_secondary_validator`		✓		
`send_message`		✓		
`set_admin`		✓	✓	✓
`set_admin_fee_share`				✓
`set_balance_ratio_min_bp`				✓
`set_bridge`				✓
`set_fee_share`				✓

Function name	Bridge	Messenger	Gas oracle	Pool
`set_gas_oracle`	✓	✓		
`set_gas_usage`	✓	✓		
`set_messenger`	✓			
`set_other_chain_ids`		✓		
`set_price`			✓	
`set_primary_validator`		✓		
`set_rebalancer`	✓			
`set_stop_authority`	✓			✓
`start_deposit`				✓
`start_swap`	✓			
`start_withdraw`				✓
`stop_deposit`				✓
`stop_swap`	✓			
`stop_withdraw`				✓
`swap`	✓			
`swap_and_bridge`	✓			
`swap_from_v_usd`				✓
`swap_to_v_usd`				✓
`withdraw`				✓
`withdraw_bridging_fee_in_tokens`	✓			
`withdraw_gas_tokens`	✓	✓		
Total (64)	**28**	**20**	**10**	**27**

B. Proposed fixes

B.1 Bridge - Insecure pattern

```
--- a/contracts/bridge/src/methods/public/swap_and_bridge.rs
+++ b/contracts/bridge/src/methods/public/swap_and_bridge.rs
@@ -34,6 +34,10 @@ pub fn swap_and_bridge(

     let token_bytes = address_to_bytes(&env, &token)?;

+    // Check that a pool is associated to token
+    let config = Bridge::get(&env)?;
+    config.pools.get(token_bytes.clone()).ok_or(Error::NoPool)?;
+
     let fee_token_amount_in_native =
         convert_bridging_fee_in_tokens_to_native_token(&env, &sender, &token,
         ↪  fee_token_amount)?;
```

B.2 Pool - Percentage input sanitization

```
--- a/contracts/pool/src/methods/admin/config_pool.rs
+++ b/contracts/pool/src/methods/admin/config_pool.rs
@@ -1,5 +1,5 @@
 use bridge_storage::*;
-use shared::{soroban_data::SimpleSorobanData, Error};
+use shared::{require, soroban_data::SimpleSorobanData, Error};
 use soroban_sdk::Env;

 use crate::storage::pool::Pool;
@@ -7,6 +7,11 @@
 pub fn set_fee_share(env: Env, fee_share_bp: u128) -> Result<(), Error> {
     Admin::require_exist_auth(&env)?;

+    require!(
+        fee_share_bp < Pool::BP,
+        Error::InvalidArg
+    );
+
     Pool::update(&env, |pool| {
         pool.fee_share_bp = fee_share_bp;
         Ok(())
@@ -16,6 +21,11 @@
 pub fn set_balance_ratio_min_bp(env: Env, balance_ratio_min_bp: u128) ->
 ↪  Result<(), Error> {
     Admin::require_exist_auth(&env)?;

+    require!(
```

```diff
+        balance_ratio_min_bp < Pool::BP,
+        Error::InvalidArg
+    );
+
     Pool::update(&env, |pool| {
         pool.balance_ratio_min_bp = balance_ratio_min_bp;
         Ok(())
@@ -25,6 +35,11 @@
 pub fn set_admin_fee_share(env: Env, admin_fee_share_bp: u128) -> Result<(),
 ↪   Error> {
     Admin::require_exist_auth(&env)?;

+    require!(
+        admin_fee_share_bp < Pool::BP,
+        Error::InvalidArg
+    );
+
     Pool::update(&env, |pool| {
         pool.admin_fee_share_bp = admin_fee_share_bp;
         Ok(())
--- a/contracts/pool/src/methods/internal/pool.rs
+++ b/contracts/pool/src/methods/internal/pool.rs
@@ -7,7 +7,7 @@
 impl Pool {

     const MAX_TOKEN_BALANCE: u128 = 2u128.pow(40);
-    const BP: u128 = 10000;
+    pub const BP: u128 = 10000;

     pub const P: u128 = 48;
     const SYSTEM_PRECISION: u32 = 3;
```