Airswift - Supply Chain Financing

Stellar Audit Bank

Reference	24-05-1650-REP
Version	1.2
Date	2024/05/24

Quarkslab SAS
10 boulevard Haussmann
75009 Paris
France

1. Project Information

Document history			
Version	Date	Details	Authors
1.0	2024/05/24	Initial version	Madigan Lebreton Elouan Wauquier
1.1	2024/06/25	Control audit #1	Madigan Lebreton Elouan Wauquier
1.2	2024/07/02	Control audit #2	Madigan Lebreton Elouan Wauquier

Quarkslab		
Contact	Role	Contact Address
Frédéric Raynal	CEO	`fraynal@quarkslab.com`
Pauline Sauder	Project Manager	`psauder@quarkslab.com`
Stavia Salomon	Sales	`ssalomon@quarkslab.com`
Madigan Lebreton	R&D Engineer	`mlebreton@quarkslab.com`
Elouan Wauquier	R&D Engineer	`ewauquier@quarkslab.com`

Airswift		
Contact	Role	Contact Address
Colin Wang	N/A	`colin.w@airswift.io`
Lize Wu	N/A	`lwu@omnisolu.com`
Yi Shi	Operations Lead	`yi.s@airswift.io`
Yan Zhang	N/A	`yan.z@airswift.io`

Contents

1 Project Information **1**

2 Executive Summary **3**
2.1 Context . . . 3
2.2 Objectives . . . 3
2.3 Disclaimer . . . 3
2.4 Findings Summary . . . 4
2.5 Recommendations and Action Plan . . . 5
2.6 Conclusion . . . 7

3 Manual Review **8**
3.1 Utility – Soroban Token . . . 8
3.2 Utility – Deployer . . . 8
3.3 Argentina – Pledge . . . 9
3.3.1 Purpose . . . 9
3.3.2 Data . . . 9
3.3.3 Code . . . 13
3.4 Argentina – Pool . . . 16
3.4.1 Purpose . . . 16
3.4.2 Data . . . 17
3.4.3 Code . . . 20
3.5 SCF – Tokenized Certificate . . . 24
3.5.1 Purpose . . . 24
3.5.2 Data . . . 25
3.5.3 Code . . . 27
3.6 SCF – Pool . . . 32
3.6.1 Purpose . . . 32
3.6.2 Data . . . 32
3.6.3 Code . . . 32

A Contract interface **36**
A.1 Argentina pledge contract interface . . . 36
A.2 Argentina pool contract interface . . . 37
A.3 Contract deployer contract interface . . . 37
A.4 Pool contract interface . . . 38
A.5 SCF Soroban contract interface . . . 39
A.6 Soroban token contract interface . . . 40

B Compilation warnings **41**

2. Executive Summary

2.1 Context

This report presents the work of the collaboration between Airswift and Quarkslab, as defined in 24-04-1622-PRO. Quarkslab's objective was to conduct a security assessment of six (6) smart contracts for Airswift's Supply Chain Financing Solution (SCF) on Soroban.

The audit parameter was defined by the content of the following GitHub repository: Airswiftio/SCF at commit `c6712721bfa685c305625bbcf2aaccd7f7c38cbd`.

On 2024/06/25, a control audit was performed to assess the status of the discovered vulnerabilities as of commit `4d65a64a3890e7e92e0d77f5d0653f2eb3f75bed`.

On 2024/07/02, a second control audit was performed to assess the status of the remaining vulnerabilities as of commit `cec7de89c8e8a2e2e19bcadeb826267d395ab918`.

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 smart contracts. To that end, Quarkslab proposed the following approach:

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

2.3 Disclaimer

This report reflects the work and results obtained within the duration of the audit for the specified scope in 24-04-1622-PRO as agreed between Airswift and Quarkslab. Tests are not guaranteed to be exhaustive and the report does not ensure that the application is bug-free.

2.4 Findings Summary

ID	Name	Fix
CRIT-1	Approvals are stored in *Instance* storage	✓
CRIT-2	Approvals are not revoked upon regular transfer	✓
CRIT-3	Approvals are stored in *Instance* storage	✓
CRIT-4	Approval is not reset during token `transfer`	✓
CRIT-5	Uncapped supply of token leads to loss of funds	✓
HIGH-1	Borrower's TC may never be transferred back after payoff, leading to loss of funds	✓
HIGH-2	Loan offer creation can be censored by front-running	✓
HIGH-3	Offer creation accepts untrusted `pool_tokens`	✓
HIGH-4	Tokenized certificate owner can split before accepting an offer	✓
MED-1	Approvals cannot be revoked	✓
MED-2	Untrusted contract call in `accept_load_offer`	✓
MED-3	Token approval can't be deleted	✓
MED-4	Offer creation accepts non-existing tokenized certificate contracts and identifiers	∼
MED-5	User may be censored through front-running	✓
LOW-1	Unbounded storage of `DataKey::FileHashes(i128)`	✓
LOW-2	Mismatched storage type of `DataKey::Owner(i128)`	✓
LOW-3	Mismatched storage type of `DataKey::Approval(ApprovalKey::ID(i128))`	✓
LOW-4	Too small type for TC amount	✓
LOW-5	Redeem time's validity is not checked at mint time	✓
LOW-6	Split may be smaller than 10% of the root's `total_amount`	✗
LOW-7	Uncapped number of verifiable credential per token	✓
INFO-1	Warnings emitted during the compilation	✓
INFO-2	Improper type for TC IDs	✓
INFO-3	Warnings emitted during the compilation	✓
INFO-4	Unused `DataKey` variants	✓
INFO-5	Fixed-point variable has limited resolution	✗
INFO-6	Bad public variable name	✓
INFO-7	Superfluous field in `Loan`	✓
INFO-8	Superfluous liquidity token	✓
INFO-9	Storage keys are not standardized	✓
INFO-10	Unused data key variants	✓

INFO-11	The `end_time` can be configured to a past timestamp	✓
INFO-12	Verifiable credential can be any format	✓

Severity: critical, high, medium, low, info
Fix status: ✗ acknowledged, ∼ mitigated, ✓ fixed

2.5 Recommendations and Action Plan

ID	Recommendations	Perimeter
CRIT-1	Move approvals to *Temporary* storage. Consider removing approvals entirely, as well as `transfer_from`, by relying on Soroban's authorization framework instead. **Control Audit (2024/07/02):** explicit approvals have been removed in favor of Soroban's authorization framework, solving the issue.	`argentina_pledge` approval
CRIT-2	Revoke approvals in `transfer` by removing the corresponding storage slot.	`argentina_pledge` approval
CRIT-3	Move approvals to *Temporary* storage.	`argentina_pledge` approval
CRIT-4	Reset the token approval in the `transfer` function. This can be done by adding `write_approval(&env, id, None);`.	`scf_soroban` transfer
CRIT-5	Consider capping the total supply of tokens. This cap must be chosen to avoid resource exhaustion in `update_and_read_expired`.	`scf_soroban` split
HIGH-1	Either add a way for the borrower to go back to the `LoanStatus::Active` state (getting their money back), or let the borrower transition from the `Paid` to `Close` state – possibly skipping the `Paid` state entirely.	`argentina_pledge` loan
HIGH-2	Generate the offer ID dynamically (e.g. sequentially) in the smart contract, and return the generated value.	`argentina_pool` loan
HIGH-3	Ensure that the `pool_token` used for creating offers is trusted. This can be done through a whitelisting mechanism.	`scf_pool`
HIGH-4	Deny `accept_offer` when the tokenized certificate is disabled.	`scf_pool`
MED-1	Allow users to revoke approvals, either by taking an `Option<Address>` as input of the `appr` function, or by using a dedicated function.	`argentina_pledge` approval
MED-2	Use a trusted registry (whitelist) of TC smart contracts.	`argentina_pool` loan

MED-3	Add a way to delete approval without overwriting with self-approving.	scf_soroban approval
MED-4	Ensure that the tc_contract and tc_id exist. tc_contract can be checked through a whitelisting mechanism. tc_id can then be checked through a call to tc_contract. **Control Audit (2024/06/25):** PARTIAL FIX. Airswift stated that they will filter out invalid offers on their front-end. TC contracts are now called at creation time, but still not verified. Thus, invalid TC contracts are possible.	scf_pool
MED-5	Implement an incremental counter handled by the contract for offer_id.	scf_pool
LOW-1	Store hashes as fixed size arrays (e.g. [u8; 32]), and add an upper bound on the length of the Vec. If the Vec cannot be bounded, store each file hash in a different slot (e.g. using a key DataKey::FileHashes(i128, u32))	argentina_pledge storage
LOW-2	When writing to DataKey::Owner(i128), store the unwrapped value if it is Some, and remove the value if it is None.	argentina_pledge storage
LOW-3	When writing to DataKey::Approval(ApprovalKey::ID(i12 store the unwrapped value if it is Some, and remove the value if it is None.	argentina_pledge storage
LOW-4	Use a larger integer type to store a TC's value, such as u64.	argentina_pledge mint
LOW-5	Check the redeem time is in the future, or explicitly allow not setting a redeem time restriction.	argentina_pledge mint
LOW-6		scf_soroban split
LOW-7	Consider capping the number of VC strings per token.	scf_soroban add_vc
INFO-1	Fix the compilation warning.	argentina_pledge
INFO-2	Use an unsigned type to store and reference TC IDs.	argentina_pledge
INFO-3	Fix the compilation warnings.	argentina_pledge
INFO-4	Remove the unused variants.	argentina_pool
INFO-5	Increase the variable resolution, e.g. by using it as a 7-decimal fixed point number. This can be done by modifying calculate_scaled_amount_with_interest. **Control Audit (2024/06/25):** ACKNOWLEDGED (as intended)	argentina_pool loan payoff
INFO-6	Change the variable name to reflect its true meaning, such as "fee", "tax", "toll", "compensation", "charge"...	argentina_pool

INFO-7	Remove the `id` field from `Loan` and modify `write_loan` accordingly.	`argentina_pool` storage
INFO-8	Remove the liquidity token and replace it with the external token.	`argentina_pool` token
INFO-9	Consider standardizing the storage key symbols by replacing the `"ORDERINFO"` symbol with a `DataKey` variant.	`scf_soroban` storage
INFO-10	Remove the unused variants.	`scf_soroban` storage
INFO-11	Consider adding checks to ensure that `end_time` is in the future.	`scf_soroban` initialize
INFO-12	Checks can be added to ensure the VC is JSON formatted. **Control Audit (2024/06/25):** SUFFICIENT MITIGATION VCs now have an upper bound on their length and their count. Considering that JSON validation is expensive on-chain and that this data is not used on-chain, we consider the mitigation sufficient.	`scf_soroban` VC

Severity: critical, high, medium, low, info

2.6 Conclusion

The audit revealed severe vulnerabilities in the codebase. We strongly advise some refactoring and going through a second audit to ensure proper remediation of the outlined issues.

Because there are similarities in the codebase, some issues are duplicates. However, we count at least 13 unique LOW or higher issues, with 2 unique CRITICAL vulnerabilities and 4 unique HIGH vulnerabilities.

Control Audit (2024/06/25): most issues have been addressed. However, 1 HIGH vulnerability was not fixed correctly, and 1 INFO issue (present in both sets of smart contracts) has been incorrectly fixed, turning it into a vulnerability rated CRITICAL (very probable denial of service). Consequently, we do not recommend the deployment of these smart contract in their current state.

Control Audit (2024/07/02): all vulnerabilities have been addressed. The remaining issues have been acknowledged by Airswift, and will be either dealt with at the front-end level (out of scope for this audit), or won't be fixed. We consider that they won't pose a security risk if properly mitigated.

3. Manual Review

The application is split in two (2) sets of smart contracts: one for the Argentinian case, and one for the general case. Although they fill similar roles, we found them to be dissimilar enough to warrant individual treatment. Consequently, some issues are found in both sets of smart contracts, while others are specific to one implementation.

The main application is made up of two (2) smart contracts:

- the Tokenized Certificate: `argentina_pledge` and `scf_soroban`, and
- the Pool: `argentina_pool` and `scf_pool`.

The Tokenized Certificate is an NFT with custom functionality, and the Pool allows users to lend their Tokenized Certificates. The general case version also uses a Deployer in `contract_deployer`, and the Argentinian case uses a generic Soroban Token in `token`,

3.1 Utility – Soroban Token

This smart contract represents a basic token and is taken straight from Stellar's `soroban-example` repository.

It is intended to be used as a liquidity token for the Argentinian implementation of the Pool.

Because of its simple nature and usage, we won't report the full analysis here and only focus on the main issues.

- The `DataKey::State(Address)` variant is unused and can be removed.
- The approval/allowance pattern is non-idiomatic on Soroban: the typical use case is to allow a smart contract to transfer tokens on a user's behalf, for example to perform a swap.

  On Soroban, the user performing the swap automatically includes a signature authorizing the transfer in their transaction. See Stellar's documentation on Soroban's Authorization Framework.
- The token metadata is stored directly using the SDK, while the remaining fields are stored using the `DataKey` enum as a key.

3.2 Utility – Deployer

The Deployer contract is used to deploy and initialize all other smart contracts.

It offers a single entrypoint `deploy_contract` and allows deploying a contract and calling multiple functions on it. This is good practice to avoid initialization front-running.

3.3 Argentina – Pledge

3.3.1 Purpose

Tokenized Certificates (TCs) constitutes the base of the Supply Chain Financing application from Airswift. They are stored and managed by this smart contract, and behave similarly to Non-Fungible Tokens (NFT).

Only the smart contract's administrator is allowed to mint new TCs, identified by an incrementally increasing ID. The TC can then be bought from the smart contract ("pledged"), transferred among users, and sold back to the smart contract ("redeemed") after some time. The TC's pledge and redeem price are the same and are paid using a set external token (e.g. USDC).

`INFO`	`INFO-1` Warnings emitted during the compilation
Perimeter	`argentina_pledge`
Fix status	✓
Description	
During the compilation, `cargo` emitted 1 warning. See Appendix B	
Recommendation	
Fix the compilation warning.	

3.3.2 Data

Instance

Key	Type	Notes
DataKey::Admin	Address	Admin-only Set in initialize
DataKey::Supply	i128	Admin-only Increment-only Unset is considered 0

DataKey::ExtToken	ExtTokenInfo {
	    address: Address,
	    decimals: u32,
	}
		Frozen
		Set in initialize

DataKey::Approval(ApprovalKey::ID(i128))	Option<Address> / Address	

DataKey::Approval(
    ApprovalKey::All(
        ApprovalAll {
            operator: Address,
            owner: Address,
        }
)
)
	bool	

The contract instance stores the administrator address (sole address allowed to mint), the count of all minted TCs (to generate the next TC ID sequentially), and the (external) token used for pledging and redeeming TCs (e.g. USDC) with its decimal count.

INFO	INFO-2 Improper type for TC IDs
Perimeter	argentina_pledge
Fix status	✓

Description

Tokenized Certificate IDs can never be negative, but are stored as `i128`. Unsigned values avoid some overhead in sign handling.

Recommendation

Use an unsigned type to store and reference TC IDs.

Approvals are typically only required for a short duration (or even a single transaction), and thus could be moved to *Temporary* storage (see the Soroban Token example).

Moreover, the approval/allowance pattern is not idiomatic on Soroban: the typical use case is to allow a smart contract to transfer tokens on a user's behalf, for example to perform a swap. On Soroban, the user performing the swap automatically includes a signature authorizing the

transfer in their transaction.
See Stellar's documentation on Soroban's Authorization Framework.

CRITICAL	CRIT-1 Approvals are stored in *Instance* storage		
Likelihood	●●●●	**Impact**	●●●●
Perimeter	`argentina_pledge` approval		
Prerequisites	None		
Fix status	✓		

Description

Approvals are typically short-lived (often a single transaction), and do not need to be kept indefinitely. Moreover, Soroban's authorization framework allows smart contracts to get direct authorization from the caller without requiring a call to `appr`.
Control Audit (2024/06/25): BAD FIX, leading to RAISED SEVERITY
Approvals were moved from *Persistent* to *Instance* storage. Either over time or through the actions of a malicious user, the accumulation of approvals makes invocations more and more expensive, until the smart contract becomes unusable (Denial of Service). See Stellar's documentation on Soroban's instance storage

Recommendation

Move approvals to *Temporary* storage. Consider removing approvals entirely, as well as `transfer_from`, by relying on Soroban's authorization framework instead.
Control Audit (2024/07/02): explicit approvals have been removed in favor of Soroban's authorization framework, solving the issue.

The *Instance* level is not appropriate for the some fields.

Persistent

Key	Type	Notes
DataKey::FileHashes(i128)	Vec<String>	
DataKey::Amount(i128)	u32	Unset is considered 0
DataKey::RedeemTime(i128)	u64	
DataKey::Owner(i128)	Option<Address> / Address	

LOW	`LOW-1` Unbounded storage of `DataKey::FileHashes(i128)`		
Likelihood	●○○○	**Impact**	●○○○
Perimeter	`argentina_pledge` storage		
Prerequisites	None		
Fix status	✓		

Description

Storage indexed by `DataKey::FileHashes(i128)` is unbounded, which can lead to high costs or Denial of Service when accessing it. The file hashes are stored as a `Vec<String>`, while hashes have a limited size.

Recommendation

Store hashes as fixed size arrays (e.g. `[u8; 32]`), and add an upper bound on the length of the `Vec`. If the `Vec` cannot be bounded, store each file hash in a different slot (e.g. using a key `DataKey::FileHashes(i128, u32)`)

LOW	`LOW-2` Mismatched storage type of `DataKey::Owner(i128)`		
Likelihood	●●●●	**Impact**	○○○○
Perimeter	`argentina_pledge` storage		
Prerequisites	None		
Fix status	✓		

Description

Values written to `DataKey::Owner(i128)` are of type `Option<Address>`, while values read from it are of type `Address`.

Recommendation

When writing to `DataKey::Owner(i128)`, store the unwrapped value if it is `Some`, and remove the value if it is `None`.

LOW	`LOW-3` Mismatched storage type of `DataKey::Approval(ApprovalKey::ID(i128))`		
Likelihood	●●●●	**Impact**	○○○○
Perimeter	`argentina_pledge` storage		
Prerequisites	None		
Fix status	✓		
Description			
Values written to `DataKey::Approval(ApprovalKey::ID(i128))` are of type `Option<Address>`, while values read from it are of type `Address`.			
Recommendation			
When writing to `DataKey::Approval(ApprovalKey::ID(i128))`, store the unwrapped value if it is `Some`, and remove the value if it is `None`.			

LOW	`LOW-4` Too small type for TC amount		
Likelihood	●○○○	**Impact**	●○○○
Perimeter	`argentina_pledge` mint		
Prerequisites			
Fix status	✓		
Description			
A TC's value ("amount") is stored as a `u32`, limiting its value to $\$2^{32} - 1 \approx \$4\,\mathrm{B}$.			
Recommendation			
Use a larger integer type to store a TC's value, such as `u64`.			

For each TC, the smart contract stores its related information in different fields indexed by the TC's ID. While this allows for cheaper storage access, it also obfuscates the natural underlying TC structure, increasing the risk of introducing bugs (such as CRIT-2)

The smart contract also allows a user to approve transfer for *all* their TCs to another address.

The *Persistent* level is appropriate for all fields.

Temporary

No *Temporary* storage is used.

3.3.3 Code

Permissioned

- `initialize` can be called only when the `DataKey::Admin` in *Instance* storage is not set.

  Anyone can call this function, but it can be called only once overall. It configures the contract's administrator and sets the external token.

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

  The administrator is used to mint new TC using `mint`, and can transfer its administrative privileges using `set_admin`.

- `set_admin` enables the administrator to transfer their privileges to another address.
- `mint` lets the administrator create new TC that users will be able to buy ("pledge"). The TC has a redeem time, which is the earliest time a user can redeem the TC (or sell it to the smart contract)

LOW	`LOW-5` Redeem time's validity is not checked at mint time		
Likelihood	●●●●	**Impact**	●○○○
Perimeter	`argentina_pledge` mint		
Prerequisites	None		
Fix status	✓		
Description			
When minting a new TC, the administrator can set the redeem time in the past, allowing users to pledge and redeem the TC at the same time.			
Recommendation			
Check the redeem time is in the future, or explicitly allow not setting a redeem time restriction.			

View

Seven (7) view functions are defined in this smart contract. These functions are permissionless, they let users retrieve information about the state of the protocol.

- `get_appr` takes a TC ID and returns which address it has been approved for, if any. It panics with `Error::NotAuthorized` if no approval has been given for this particular TC, or if the TC hasn't been minted yet.
- `is_appr` takes a pair of addresses and returns whether the first one has approved the second one to transfer any of its TC on its behalf.
- `get_amount` takes a TC ID and returns its price (for both `pledge` and `redeem`). It returns 0 if the TC hasn't been minted yet.

- `get_owner` takes a TC ID and returns its current owner (can be the smart contract if it has not yet been pledged, a regular user, or `None` if it has been redeemed). It panics with `Error::NotFound` if the TC hasn't been minted yet.
- `get_file_hashes` takes a TC ID and returns the list of file hashes it has been associated with at mint time. It panics with `Error::NotFound` if the TC hasn't been minted yet.
- `get_redeem_time` takes a TC ID and returns the earliest time it can be redeemed. It panics with `Error::NotFound` if the TC hasn't been minted yet.
- `get_ext_token` returns the external token configured at initialize time along with its decimal count. It panics if the contract has not been initialized yet.

User

Users can `pledge` and `redeem` TCs minted by the administrator (in this order). While a TC has been pledged, users can transfer it like a normal NFT using `transfer`, `transfer_from`, `appr` and `appr_all`.

- `pledge` transfers a freshly minted TC (i.e. owned by the smart contract) to the authorizing user, against the TC's `amount` number of external tokens.
- `redeem` burns a pledged TC (i.e. transfers it to `None`) belonging to the authorizing user. Upon doing so, the authorizing user is paid the burnt TC's `amount` number of external tokens. As a given TC's `amount` cannot be updated, this value is the same as the one in `pledge`.
- `appr` gives the authorizing user's approval to an address (typically a smart contract such as a pool) to transfer a given TC they own on their behalf, using `transfer_from`. A new approval overrides any previous approval of this type.
- `appr_all` gives the authorizing user's approval to an address (typically a smart contract such as a pool) to transfer any TC they own on their behalf, using `transfer_from`. A new approval does not override approvals for other addresses.
- `transfer` transfers a TC from the authorizing user (if they own the TC) to an address, ignoring approval rules. This function does not reset the approval given for the transferred TC.
- `transfer_from` transfers a TC from one user (if they own the TC) to an address it they approved it before (using `appr` or `appr_all`). This function does reset the approval given for the transferred TC.

MEDIUM	`MED-1` Approvals cannot be revoked		
Likelihood	●●●○	**Impact**	●○○○
Perimeter	`argentina_pledge` approval		
Prerequisites	User called `appr`		
Fix status	✓		

Description

There is no dedicated functionality to revoke approvals, so the only way to revoke an approval is by give it to a bogus address.

Recommendation

Allow users to revoke approvals, either by taking an `Option<Address>` as input of the `appr` function, or by using a dedicated function.

CRITICAL	`CRIT-2` Approvals are not revoked upon regular transfer		
Likelihood	●●●●	**Impact**	●●●●
Perimeter	`argentina_pledge` approval		
Prerequisites	User called `appr` , then `transfer`		
Fix status	✓		

Description

Approval set by calling `appr` on a given TC is not revoked when the TC is transferred using `transfer` . This enables an attacker to configure an approval on themselves, lend the TC to a victim in exchange for liquidity tokens, and transfer the TC back to themselves without paying off their loan.

Recommendation

Revoke approvals in `transfer` by removing the corresponding storage slot.

3.4 Argentina – Pool

3.4.1 Purpose

The pool enables users to loan other users TCs for a fee (“rate”).

Loan management is handled by this smart contract and paced by the loan’s status.

Users first propose to loan another user’s TC at the current configured rate. The TC is returned when both the borrower has paid its debt back, and the creditor accepts to close the loan.

Transactions are performed using a pool liquidity token that can be exchanged at a rate of 1 : 1 with the external token (e.g. USDC).

INFO	`INFO-3` Warnings emitted during the compilation
Perimeter	`argentina_pledge`
Fix status	✓
Description	
During the compilation, `cargo` emitted 5 warnings.	
Recommendation	
Fix the compilation warnings.	

3.4.2 Data

All data keys are variants of `DataKey`.

Storage is accessed through the `DatKey` enum:

```
pub enum DataKey {
    Admin,
    ExtToken,
    PoolToken,
    Supply,
    Amount(i128),
    RedeemTime(i128),
    Owner(i128),
    RatePercent,
    Loan(i128),
}
```

Four (4) of these variants are never used:

- `DataKey::Supply` (used at the time of the control audit),
- `DataKey::Amount(i128)`,
- `DataKey::RedeemTime(i128)`, and
- `DataKey::Owner(i128)`.

INFO	`INFO-4` Unused `DataKey` variants
Perimeter	`argentina_pool`
Fix status	✓
Description	
Four (4) variants of `DataKey` are never used.	
Recommendation	
Remove the unused variants.	

Instance

Key	Type	Notes
`DataKey::Admin`	`Address`	Admin-only Set in `initialize`
`DataKey::RatePercent`	`u32`	Admin-only Set in `initialize` Unset is considered 0, but can never happen

`DataKey::PoolToken`	`TokenInfo {`
	`address: Address,`
	`decimals: u32,`
	`}`
		Frozen
		Set in `initialize`

`DataKey::ExtToken`	`TokenInfo {`
	`address: Address,`
	`decimals: u32,`
	`}`
		Frozen
		Set in `initialize`

The contract instance stores the administrator address (sole address allowed to set the global rate), the global rate (as a percentage of the base price), the pool token used as a liquidity token for loans, and the (external) token used for minting and burning liquidity tokens.

INFO	`INFO-5` Fixed-point variable has limited resolution
Perimeter	`argentina_pool` loan payoff
Fix status	✗

Description

The value stored at `DataKey::RatePercent` is used as a 2-decimal fixed point number. This limits its resolution to 1%, while its upper bound is unrealistically high at $\frac{2^{32}-1}{100}$.

Recommendation

Increase the variable resolution, e.g. by using it as a 7-decimal fixed point number. This can be done by modifying `calculate_scaled_amount_with_interest`.

Control Audit (2024/06/25): ACKNOWLEDGED (as intended)

INFO	`INFO-6` Bad public variable name
Perimeter	`argentina_pool`
Fix status	✓

Description

A "rate" is a ratio between two quantities, most often one quantity with respect to time. In this context, the fee added is fixed.

Recommendation

Change the variable name to reflect its true meaning, such as "fee", "tax", "toll", "compensation", "charge"...

The *Instance* level is appropriate for the configured fields.

Persistent

Key	Type	Notes
`DataKey::Loan(i128)`	`Loan { id: i128, borrower: Address, creditor: Address, amount: i128, tc_address: Address, tc_id: i128, rate_percent: u32, status: LoanStatus, }`	

Each potential loan is stored using a dedicated structure, in a dedicated storage slot indexed by offer ID. Thus, there can be several loans for a single TC.

INFO	`INFO-7` Superfluous field in `Loan`
Perimeter	`argentina_pool` storage
Fix status	✓

Description

The `id` field in `Loan` is superfluous and takes up persistent storage space. In every occurrence of its usage, its value is available elsewhere.

Recommendation

Remove the `id` field from `Loan` and modify `write_loan` accordingly.

The *Persistent* level is appropriate for this field.

Temporary

No *Temporary* storage is used.

3.4.3 Code

Permissioned

- The `initialize` function can be called only when the `DataKey::Admin` in *Instance* storage is not set.

  Anyone can call this function, but it can be called only once overall. It configures the contract's administrator, creates and initializes a standard token intended to be used as the pool's liquidity token, sets the external token, and the initial loan payoff fee.

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

  The administrator is used to update the pool's loan payoff fee using `set_rate`, and can transfer its administrative privileges using `set_admin`.

  The liquidity token can be exchanged at 1 : 1 with the external token using `deposit` and `withdraw`. Its administrator is the pool smart contract itself.

  Both tokens' address cannot be updated.

- The `set_admin` function enables the administrator to transfer their privileges to another address.
- The `set_rate` function lets the administrator modify the loan payoff fee.

View

Ten (10) view functions are defined in this smart contract. These functions are permissionless, they let users retrieve information about the state of the protocol.

- `get_loan_status` takes a loan ID and returns its status (`Pending`, `Active`, `Paid`, or `Closed`). It panics with `Error::NotFound` if the loan has not been created yet, or if the contract hasn't been initialized yet.
- `get_loan_tc` takes a loan ID and returns its associated TC address and ID. It panics with `Error::NotFound` if the loan has not been created yet, or if the contract hasn't been initialized yet.
- `get_loan_borrower` takes a loan ID and returns its borrower. It panics with `Error::NotFound` if the loan has not been created yet, or if the contract hasn't been initialized yet.

- `get_loan_creditor` takes a loan ID and returns its creditor. It panics with `Error::NotFound` if the loan has not been created yet, or if the contract hasn't been initialized yet.
- `get_loan_amount` takes a loan ID and returns its price, excluding fee, in the pool's liquidity token's base unit. It panics with `Error::NotFound` if the loan has not been created yet, or if the contract hasn't been initialized yet.
- `get_payoff_amount` takes a loan ID and returns its payoff price, including fee, scaled by the pool's liquidity token's decimals and rounded down. It panics with `Error::NotFound` if the loan has not been created yet, or if the contract hasn't been initialized yet.
- `get_loan_rate` takes a loan ID and returns its associated payoff fee (corresponding to the pool's payoff fee when the offer was created). It panics with `Error::NotFound` if the loan has not been created yet, or if the contract hasn't been initialized yet.
- `get_pool_rate` returns the pool's current payoff fee, or 0 if the contract has not been initialized yet.
- `get_ext_token` returns the address of the external token (e.g. USDC). It panics if the smart contract hasn't been initialized yet.
- `get_liquidity_token` returns the address of the pool's liquidity token. It panics if the smart contract hasn't been initialized yet.

User

Users first need to exchange external tokens for the pool's liquidity tokens, at a rate of 1 : 1, using `deposit` and `withdraw`.

- `deposit` takes an amount, transfers it from the authorized user (`from`) to the smart contract, and mints that amount of liquidity token to the authorized user.
- `withdraw` takes an amount, transfers it from the smart contract to the authorized user (`from`), and burns that amount of liquidity token from the authorized user.

Because there is no restriction on deposits and withdrawals, because the exchange rate is fixed at 1 : 1, and because no additional administrative rights are provided to the pool, the liquidity token is superfluous and can be replaced with the external token at every place. This reduces the attack surface and simplifies the interactions with the smart contract.

INFO	INFO-8 Superfluous liquidity token
Perimeter	argentina_pool token
Fix status	✓

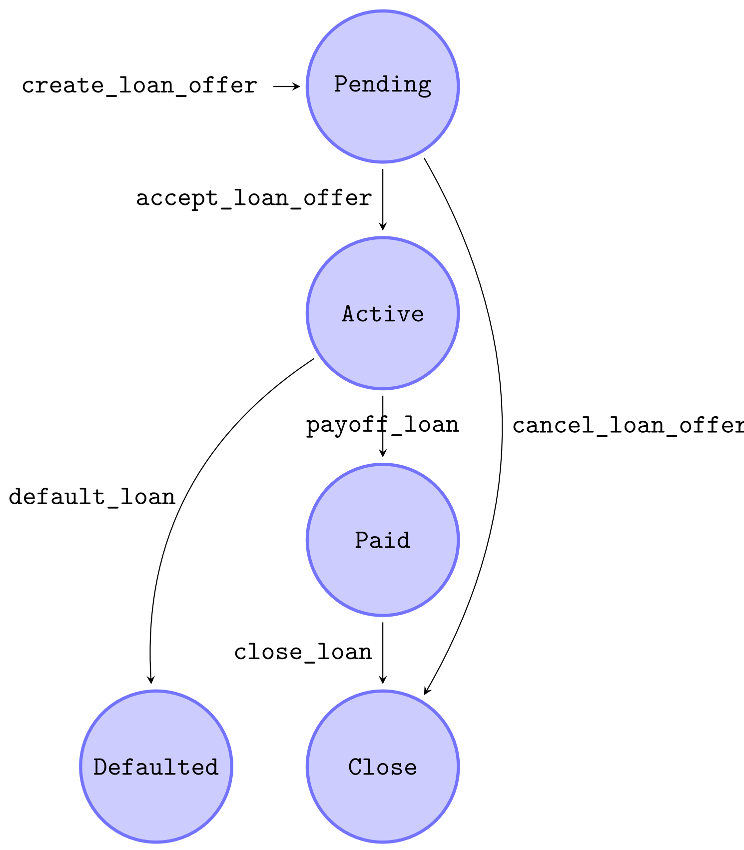
Description

The liquidity token is a standard token created by and for the pool. It can be exchanged at any time at a rate of 1 : 1 with the external token. The pool only performs basic mint and burn operations. Thus, the token is superfluous and can be replaced with the external token directly.

Recommendation

Remove the liquidity token and replace it with the external token.

Then, they can create and interact with loans using the following five (5) functions presented in a state machine. We verified that each function can only be called when the loan is in the correct state, and that the function updates the state according to the state machine.



HIGH	`HIGH-1` Borrower's TC may never be transferred back after payoff, leading to loss of funds		
Likelihood	●●●○	**Impact**	●●●○
Perimeter	`argentina_pledge` loan		
Prerequisites	User paid off their loan		
Fix status	✓		

Description

Opening and closing a loan requires both users' synchronicity: to open a loan, a creditor needs to create an offer that the borrower needs to accept. The creditor can walk back using `cancel_loan_offer` if they change their mind. When closing a loan, the borrower pays their debt back with an added fee, but the creditor is not compelled to transfer the TC back to the borrower (e.g. if the TC appreciates in value). In this case, the creditor does not get their money back but gets to keep the TC, while the borrower loses the TC, the borrowed money and the fee.

Control Audit (2024/06/25): NOT FIXED

Airswift modified the behavior but did not fully fix the issue. When the loan is accepted, the TC is transferred to the smart contract instead of the creditor. Later, the admin is able to transition an `Active` loan to the `Defaulted` final state, transferring the TC from the smart contract to the creditor. This state is equivalent to the `Active` state before the change (the creditor owns the TC), except the loan cannot be paid back anymore. If a loan is in the `Paid` state, a dissident creditor can refuse to close the loan, leading to the original problematic situation discribed above, with the exception of the creditor not keeping the TC.

Control Audit (2024/07/02): FIXED

Airswift merged the `Paid` and `Close` state by removing the `close_loan` function, and making `payoff_loan` transition to the `Close` state.

Recommendation

Either add a way for the borrower to go back to the `LoanStatus::Active` state (getting their money back), or let the borrower transition from the `Paid` to `Close` state – possibly skipping the `Paid` state entirely.

- `create_loan_offer` creates a loan offer on an existing TC by specifying its address and ID. This is meant to be used on other people's TC to offer them liquidity in exchange for their TC. To create the offer, the authorized user (creditor) must transfer the TC's `amount` of liquidity tokens to the pool, and this amount is saved in the loan structure. The caller specifies the offer ID, and the contract verifies that an offer with the given ID does not already exist. The TC address is arbitrary and is called in `create_loan_offer`, `accept_loan_offer`, and `close_loan`.
- `cancel_loan_offer` if the offer was not accepted, the original creditor can cancel their offer. They are reimbursed the amount they initially transferred (no call to the TC).
- `accept_loan_offer` if an offer exists for a TC they own, a user can accept the offer and receive the TC's `amount` of liquidity tokens. The TC is transferred to the creditor (with a call to the saved `tc_address`).

- `payoff_loan` if the authorized user is the borrower, transfers the TC's `amount` of liquidity tokens to the smart contract with the added fee.
- `close_loan` if the authorized user is the creditor, transfers the TC's `amount` of liquidity tokens to the creditor with the added fee, and transfers the TC back to the borrower.

HIGH	`HIGH-2` Loan offer creation can be censored by front-running		
Likelihood	●●●○	**Impact**	●●○○
Perimeter	`argentina_pool` loan		
Prerequisites	Visibility into the mempool		
Fix status	✓		
Description			
A malicious user can prevent the creation of a loan offer by front-running it with the creation of a dummy loan offer with the same ID. For example, this enables them to either censor a particular user, or to prevent any user from posting a loan offer for a specific TC.			
Recommendation			
Generate the offer ID dynamically (e.g. sequentially) in the smart contract, and return the generated value.			

MEDIUM	`MED-2` Untrusted contract call in `accept_load_offer`		
Likelihood	●○○○	**Impact**	●●●●
Perimeter	`argentina_pool` loan		
Prerequisites	Social Engineering		
Fix status	✓		
Description			
A malicious user can publish a loan offer with an arbitrary TC. If a user accepts this offer, the TC's `transfer` function is called which can in turn perform arbitrary operations, such as transferring the victim's funds to the attacker.			
Recommendation			
Use a trusted registry (whitelist) of TC smart contracts.			

3.5 SCF – Tokenized Certificate

3.5.1 Purpose

The Tokenized Certificate smart contract is defined in the `scf_soroban` directory of the repository. This contract defines tokenized certificates that represent an amount of an external token. A tokenized certificate can be split into multiple ones with portions of the original amount before being disabled.

Every tokenized certificate share a single root tokenized certificate created by the administrator of the smart contract.

3.5.2 Data

Instance

- a token order information structure, with the symbol `"ORDERINFO"`;
- the administrator address, with the symbol `DataKey::Admin`;
- an approval boolean for each tokenized certificate identifier, with the symbol `DataKey::Approval(ApprovalKey::ID(id))`;
- an approval boolean for mapping two addresses, with the symbol `DataKey::Approval(ApprovalAll { operator, owner })`;

The `TokenOrderInfo` structure stores external token's specific data. Three fields are defined in this specific structure:

- `buyer_address` : the address that can pay the amount of external token;
- `total_amount` : the total amount of external token to pay;
- `end_time` : the deadline after which tokenized certificates can be marked as expired.

INFO	`INFO-9` Storage keys are not standardized
Perimeter	`scf_soroban` storage
Fix status	✓

Description

The storage keys in the Tokenized Certificate contract are initialized with a variant of the `DataKey` enumeration as symbol, except for the order information that is initialized with the `"ORDERINFO"` string as symbol.

Recommendation

Consider standardizing the storage key symbols by replacing the `"ORDERINFO"` symbol with a `DataKey` variant.

CRITICAL	`CRIT-3` Approvals are stored in *Instance* storage		
Likelihood	●●●●	**Impact**	●●●●
Perimeter	`argentina_pledge` approval		
Prerequisites	None		
Fix status	✓		

Description

Approvals are typically short-lived (often a single transaction), and do not need to be kept indefinitely. Moreover, Soroban's authorization framework allows smart contracts to get direct authorization from the caller without requiring a call to `appr` .

Control Audit (2024/06/25): BAD FIX, leading to RAISED SEVERITY
Approvals were moved from *Persistent* to *Instance* storage. Either over time or through the actions of a malicious user, the accumulation of approvals makes invocations more and more expensive, until the smart contract becomes unusable (Denial of Service). See Stellar's documentation on Soroban's instance storage

Control Audit (2024/07/02): explicit approvals have been removed in favor of Soroban's authorization framework, solving the issue.

Recommendation

Move approvals to *Temporary* storage.

The *Instance* level is not appropriate for some configured fields.

Persistent

- the owner of a tokenized certificate identifier, with the symbol `DataKey::Owner(id)`;
- the recipient of a tokenized certificate identifier, with the symbol `DataKey::Recipient(id)`;
- the "verifiable credential" of a tokenized certificate identifier, with the symbol `DataKey::VC(id)`;
- the sub tokenized certificates of a tokenized certificate when split is used, with the symbol `DataKey::SubTCInfo(id)`;
- a boolean for each token identifier indicating if it is disabled, with the symbol `DataKey::Disabled(id)`;
- the token total supply as a signed integer, with the symbol `DataKey::Supply`;
- the external token address and decimals, with the symbol `DataKey::ExternalToken`;
- a boolean for the expiration of tokenized certificates, with the symbol `DataKey::Expired`;
- a boolean for the pay status, with the symbol `DataKey::Paid`.

INFO	`INFO-10` Unused data key variants
Perimeter	`scf_soroban` storage
Fix status	✓
Description	
Four (4) variants of `DataKey` are never used.	
Recommendation	
Remove the unused variants.	

The *Persistent* level is appropriate for the configured fields.

Temporary

No *Temporary* storage is used.

3.5.3 Code

Permissioned

The administrator is responsible for initializing the contract through the `initialize` function. This initialization sets three important parameters:

- `buyer_address` : The address that will pay the amount of external token;
- `total_amount` : The total amount of external token to be paid;
- `end_time` : The deadline after which tokenized certificates can be marked as expired.

INFO	`INFO-11` The `end_time` can be configured to a past timestamp
Perimeter	`scf_soroban` initialize
Fix status	✓
Description	
The initialization of the Tokenized Certificate contract allows setting the end timestamp to the past.	
Recommendation	
Consider adding checks to ensure that `end_time` is in the future.	

Then, the administrator has exclusive access to four functions:

- `mint_original` : mints the root tokenized certificate;
- `burn` : allows administrator to burn any tokenized certificate;

- `set_external_token_provider` : configures the external token address;
- `add_vc` : adds a verifiable credential to a tokenized certificate.

View

The smart contract provides several functions to read the contract state. These functions are:

- `admin` : retrieves the administrator address;
- `get_appr` : gets the approved address of a token identifier;
- `is_appr` : retrieves a boolean indicating the state of approval of an address (spender) on another address;
- `amount` : retrieves the amount attached to the input token identifier;
- `parent` : retrieves the parent token identifier attached to the input token identifier;
- `owner` : retrieves the owner of the input token identifier;
- `vc` : retrieves the verifiable credential attached to the input token identifier;
- `get_all_owned` : retrieves all the tokens owned by an address;
- `is_disabled` : retrieves the disabled status of the input token identifier;
- `check_paid` : retrieves the boolean indicating if amount has been paid by buyer;
- `recipient` : retrieves the recipient of a token identifier, it is used during split.

User

The smart contract provides several functions to users. These functions are:

- `appr` : approves an address on a tokenized certificate;
- `appr_all` : approves an address to transfer any token owned by another address;
- `transfer` : transfers tokenized certificate from its owner;
- `transfer_from` : transfers tokenized certificate from its owner by an approved address;
- `split` : splits a tokenized certificate into sub certificates, disabling the split token;
- `redeem` : burns a tokenized certificate and transfer the amount of external token that this certificate holds;
- `sign_off` : allows a recipient to gain ownership on a tokenized certificate after a split;
- `pay_off` : used by the buyer to pay the total amount of external token.

Token owners are able to set approval on a specific token that they owned through the `appr` function. They can also approve an address for all the token they hold through the `appr_all` function. An approved address can then transfer the token through `transfer_from` function, which will also reset the approval on token.

However, when a transfer is made through `transfer` , token approval is not reset to its default value. This allows the owner to approve himself on the token, transfer it to another

entity and transfer it back through to himself `transfer_from`. Moreover, as accepting an offer in the Offer Pool contract uses the `transfer` function, a malicious token owner can accept an offer that will transfer the token to the offerer and transfer it back to himself through this issue.

CRITICAL	`CRIT-4` Approval is not reset during token `transfer`		
Likelihood	●●●●	**Impact**	●●●●
Perimeter	`scf_soroban` transfer		
Prerequisites	Ownership of a token		
Fix status	✓		

Description

An address can be set as approved on a token identifier, allowing it to transfer the token through `transfer_from`. The `transfer` function doesn't reset this approval during a transfer, allowing the owner of the token to transfer it and later retrieve it.

Recommendation

Reset the token approval in the `transfer` function. This can be done by adding `write_approval(&env, id, None);`.

MEDIUM	`MED-3` Token approval can't be deleted		
Likelihood	●●●○	**Impact**	●○○○
Perimeter	`scf_soroban` approval		
Prerequisites	Ownership of a token		
Fix status	✓		

Description

The approval set for a given token can't be deleted, it can only be overwritten. This forces owners to approve their self to delete an approval.

Recommendation

Add a way to delete approval without overwriting with self-approving.

A tokenized certificate can be split into multiple others. For example, a tokenized certificate with amount 1000 can be split into three certificates with amount 500, 300 and 200. Splitting a token will increase the total supply of tokenized certificates. A single token can be split into an infinite number of other tokens. The `update_and_read_expired` function is used in most contract entry-points. When the end time is reached, this function iterates through all the tokens to mark them as expired. However, a large total supply will lead this function to high resource consumption, breaking the whole protocol. Users will not be able to redeem their tokenized certificates for the amount of external token, and funds will be lost.

CRITICAL	`CRIT-5` Uncapped supply of token leads to loss of funds		
Likelihood	●●●○	**Impact**	●●●●
Perimeter	`scf_soroban` split		
Prerequisites	Ownership of a token		
Fix status	✓		

Description

An owner of a tokenized certificate can split it into an infinite number of tokens, leading to denial of service of the protocol.
This is due to a lack of supply capping and the fact that splitting with zero amount is possible.
Control Audit (2024/06/25):
Each split now must be 10% of the *root*'s `total_amount`, and there can be a depth of at most 5 splits.

Recommendation

Consider capping the total supply of tokens. This cap must be chosen to avoid resource exhaustion in `update_and_read_expired`.

LOW	`LOW-6` Split may be smaller than 10% of the root's `total_amount`		
Likelihood	●●●○	**Impact**	●○○○
Perimeter	`scf_soroban` split		
Prerequisites	Ownership of a token		
Fix status	✗		

Description

If the sum of the split token is less than the parent token, an addition child is created with the remaining value, but this value can be less than 10% of the root's `total_amount`.
Because splits are limited in depth, and because only one small sub-TC may be created per split, this issue is classified as LOW.

Recommendation

"Verifiable Credential" strings are attached to every tokenized certificate. The administrator can attach VC strings to any token. The number of VC string per token is not capped. This string is supposed to be formatted in JSON but can be any format.

LOW	`LOW-7` Uncapped number of verifiable credential per token		
Likelihood	●○○○	**Impact**	●○○○
Perimeter	`scf_soroban` add_vc		
Prerequisites	Administrator		
Fix status	✓		

Description

The administrator can add an infinite number of VC strings to a token.
This can lead to resource exhaustion when retrieving these strings through the `vc` view entrypoint.

Recommendation

Consider capping the number of VC strings per token.

INFO	`INFO-12` Verifiable credential can be any format
Perimeter	`scf_soroban` VC
Fix status	✓

Description

The verifiable credential attached to a tokenized certificate can be any format.

Recommendation

Checks can be added to ensure the VC is JSON formatted.
Control Audit (2024/06/25): SUFFICIENT MITIGATION
VCs now have an upper bound on their length and their count. Considering that JSON validation is expensive on-chain and that this data is not used on-chain, we consider the mitigation sufficient.

3.6 SCF – Pool

3.6.1 Purpose

The Offer Pool smart contract is defined in the `scf_pool` directory of the repository. This contract allows users to create offers for tokenized certificates. Then, owners of token can accept an offer in exchange for his token.

3.6.2 Data

Instance

- the administrator address, with the symbol `DataKey::Admin`;
- a WASM hash for the pool token contract, with the symbol `ContractDataKey::PoolTokenWasmHash`;
- a map of address to address, with the symbol `ContractDataKey::PoolTokens`;
- an "external token" address for each pool token created, with the symbol `ContractDataKey::ExtToken(pool`

The *Instance* level is appropriate for the configured fields.

Persistent

- an offer structure per offer identifier, with the symbol `DataKey::Offer(offer_id)`;

The offer structure is defined as follows.

```
pub struct Offer {
  pub from: Address,
  pub pool_token: Address,
  pub amount: i128,
  pub tc_contract: Address,
  pub tc_id: i128,
  pub status: i128,
}
```

The *Persistent* level is appropriate for the configured fields.

3.6.3 Code

Permissioned

The administrator is responsible for initializing the contract through the `initialize` function. This initialization sets two important parameters:

- `admin` : The administrator address;

- `token_wasm_hash` : The WASM hash used to deploy pool tokens.

Then, the administrator has exclusive access to four functions:

- `set_admin` ;
- `add_pool_token` ;
- `expire_offer` .

View

The Offer Pool contract provides several functions to read the contract state. These functions are:

- `admin` ;
- `get_pool_tokens` ;
- `get_offer` ;
- `get_ext_token` .

User

The smart contract provides several functions to users. These functions are:

- `deposit` : allows a user to deposit an amount of external tokens to receive an equivalent amount of pool tokens.
- `withdraw` : allows a user to burn an amount of pool tokens to receive an equivalent amount of external tokens.
- `create_offer` : allows a user to create an offer. This offer proposes an amount of a specified token for a Tokenized Certificate. The amount of token is transferred to the contract.
- `expire_offer` : is used by the administrator or the offerer to cancel an offer. Canceling an offer will send back the amount of token to the offerer.
- `accept_offer` : called by the owner of the Tokenized Certificate. It transfers the offered amount to the owner, and the Tokenized Certificate is transferred to the offerer.

The `accept_offer` function accepts any `pool_token` address for creating an offer. This allows an attacker to create a malicious token and create an offer with it. Then, two scenarios can appear:

- the administrator cancels this malicious offer;
- the tokenized certificate owner accepts this malicious offer.

In both cases, the caller will interact with the malicious token. By the inner working of the Soroban platform, this interaction with the malicious contract will allow draining all tokens from this user because authorization will be given (bypass of `require_auth()`).

HIGH	`HIGH-3` Offer creation accepts untrusted `pool_tokens`		
Likelihood	●●○○	**Impact**	●●●●
Perimeter	`scf_pool`		
Prerequisites	Accept or cancel an offer		
Fix status	✓		

Description

An attacker can create offer with malicious token. During `accept_offer` or `cancel_offer`, an external call to this malicious token will be executed, leading to potential drain of users' funds.
At the time of the control audit, `pool_token` has been renamed `ext_token` in the function body, and is checked against a whitelist.

Recommendation

Ensure that the `pool_token` used for creating offers is trusted. This can be done through a whitelisting mechanism.

Moreover, offers can be created for non-existing tokenized certificate contracts and identifiers.

MEDIUM	`MED-4` Offer creation accepts non-existing tokenized certificate contracts and identifiers		
Likelihood	●●○○	**Impact**	●●○○
Perimeter	`scf_pool`		
Prerequisites			
Fix status	∼		

Description

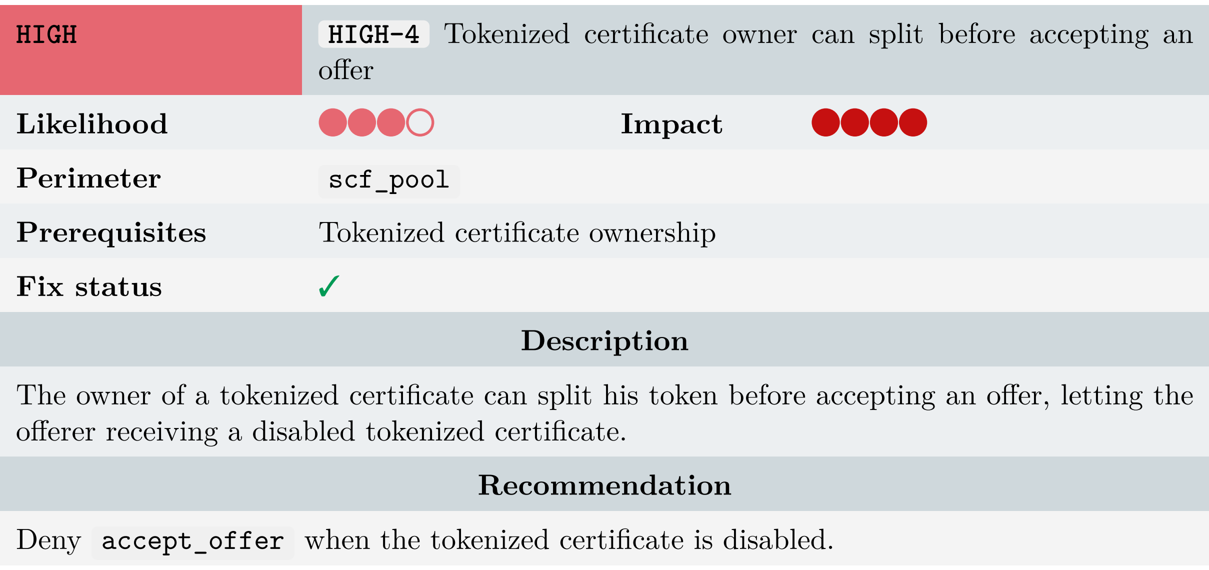
A user can create offers for non-existing TC contracts and non-existing token identifiers

Recommendation

Ensure that the `tc_contract` and `tc_id` exist. `tc_contract` can be checked through a whitelisting mechanism. `tc_id` can then be checked through a call to `tc_contract`.
Control Audit (2024/06/25): PARTIAL FIX. Airswift stated that they will filter out invalid offers on their front-end.
TC contracts are now called at creation time, but still not verified. Thus, invalid TC contracts are possible.

The purpose of Offer Pool contract is to create offers for tokenized certificates. These tokenized certificates are valuable because they hold an amount of tokens that will be paid by the `buyer_address`. However, an owner of a tokenized certificate can both accept an offer and keep this amount of token held by the certificate. This can be done through a split of the certificate into a single child certificate before accepting an offer. The offerer will receive a disabled tokenized certificate, and the owner will receive the offered amount of tokens and will

have a newly created tokenized certificate with its parent value.



HIGH	HIGH-4 Tokenized certificate owner can split before accepting an offer		
Likelihood	●●●○	**Impact**	●●●●
Perimeter	`scf_pool`		
Prerequisites	Tokenized certificate ownership		
Fix status	✓		
Description			
The owner of a tokenized certificate can split his token before accepting an offer, letting the offerer receiving a disabled tokenized certificate.			
Recommendation			
Deny `accept_offer` when the tokenized certificate is disabled.			

When creating an offer, an `offer_id` is passed as input. This identifier must be unique and unused. An attacker can leverage this to front-run a legit user's transaction to censor him.

MEDIUM	MED-5 User may be censored through front-running		
Likelihood	●○○○	**Impact**	●●●○
Perimeter	`scf_pool`		
Prerequisites			
Fix status	✓		
Description			
The `offer_id` parameter passed as input must be unique and not used. An attacker can use this to censor a user by creating 0 amount offers with the same identifier through front-running.			
Recommendation			
Implement an incremental counter handled by the contract for `offer_id`.			

A. Contract interface

A.1 Argentina pledge contract interface

External function	Admin-only	Operations
`initialize`	✗	Read/Write
`set_admin`	✓	Read/Write
`mint`	✓	Read/Write
`transfer`	✗	Read/Write
`transfer_from`	✗	Read/Write
`appr`	✗	Read/Write
`appr_all`	✗	Read/Write
`get_appr`	✗	Read-Only
`is_appr`	✗	Read-Only
`pledge`	✗	Read/Write
`redeem`	✗	Read/Write
`get_amount`	✗	Read-only
`get_owner`	✗	Read-only
`get_file_hashes`	✗	Read-only
`get_ext_token`	✗	Read-only
`get_redeem_time`	✗	Read-only

A.2 Argentina pool contract interface

External function	Admin-only	Operations
`initialize`	✗	Read/Write
`set_admin`	✓	Read/Write
`set_rate`	✓	Read/Write
`deposit`	✗	Read/Write
`withdraw`	✗	Read/Write
`create_loan_offer`	✗	Read/Write
`cancel_loan_offer`	✗	Read/Write
`accept_loan_offer`	✗	Read/Write
`payoff_loan`	✗	Read/Write
`close_loan`	✗	Read/Write
`get_loan_rate`	✗	Read-Only
`get_pool_rate`	✗	Read-Only
`get_loan_tc`	✗	Read-Only
`get_loan_borrower`	✗	Read-Only
`get_loan_creditor`	✗	Read-Only
`get_liquidity_token`	✗	Read-Only
`get_ext_token`	✗	Read-Only
`get_payoff_amount`	✗	Read-Only
`get_loan_amount`	✗	Read-Only
`get_loan_status`	✗	Read-Only

A.3 Contract deployer contract interface

External function	Admin-only	Operations
`deploy_contract`	✗	Read/Write

A.4 Pool contract interface

External function	Admin-only	Operations
`initialize`	✗	Read/Write
`admin`	✗	Read-Only
`set_admin`	✓	Read/Write
`add_pool_token`	✓	Read/Write
`get_pool_tokens`	✗	Read-Only
`deposit`	✗	Read/Write
`withdraw`	✗	Read/Write
`create_offer`	✗	Read/Write
`expire_offer`	✗	Read/Write
`get_offer`	✗	Read-Only
`accept_offer`	✗	Read/Write
`get_ext_token`	✗	Read-Only

A.5 SCF Soroban contract interface

External function	Admin-only	Operations
`initialize`	✗	Read/Write
`admin`	✗	Read-Only
`set_admin`	✓	Read/Write
`appr`	✗	Read/Write
`appr_all`	✗	Read/Write
`get_appr`	✗	Read-Only
`is_appr`	✗	Read-Only
`amount`	✗	Read-Only
`parent`	✗	Read-Only
`owner`	✗	Read-Only
`vc`	✗	Read-Only
`get_all_owned`	✗	Read-Only
`is_disabled`	✗	Read-Only
`transfer`	✗	Read/Write
`transfer_from`	✗	Read/Write
`mint_original`	✓	Read/Write
`burn`	✓	Read/Write
`split`	✗	Read/Write
`redeem`	✗	Read/Write
`set_external_token_provider`	✓	Read/Write
`check_paid`	✗	Read-Only
`check_expired`	✗	Read-Only
`recipient`	✗	Read-Only
`sign_off`	✗	Read/Write
`pay_off`	✗	Read/Write
`add_vc`	✓	Read/Write

A.6 Soroban token contract interface

External function	Admin-only	Operations
`initialize`	✗	Read/Write
`mint`	✓	Read/Write
`set_admin`	✓	Read/Write
`allowance`	✗	Read/Write
`approve`	✗	Read/Write
`balance`	✗	Read/Write
`transfer`	✗	Read/Write
`transfer_from`	✗	Read/Write
`burn`	✗	Read/Write
`burn_from`	✗	Read/Write
`decimals`	✗	Read-Only
`name`	✗	Read-Only
`symbol`	✗	Read-Only

B. Compilation warnings

```
   Compiling argentina-pledge v0.1.0 (soroban/argentina_pledge)
warning: unused import: `String`
 --> src/storage_types.rs:1:42
  |
1 | use soroban_sdk::{contracttype, Address, String};
  |                                          ^^^^^^
  |
  = note: `#[warn(unused_imports)]` on by default

...

   Compiling argentina-pool v0.1.0 (soroban/argentina_pool)
warning: unused import: `Symbol`
  --> src/contract.rs:13:90
   |
13 |     contract, contractimpl, panic_with_error, token, vec, Address, BytesN, Env,
↪  IntoVal, Symbol,
   |
   ↪   ^^^^^^
   |
   = note: `#[warn(unused_imports)]` on by default

warning: unused import: `String`
 --> src/interface.rs:1:41
  |
1 | use soroban_sdk::{Address, BytesN, Env, String};
  |                                         ^^^^^^

warning: unused import: `String`
 --> src/storage_types.rs:1:42
  |
1 | use soroban_sdk::{contracttype, Address, String};
  |                                          ^^^^^^

warning: unused variable: `from`
   --> src/contract.rs:213:28
    |
213 |     fn payoff_loan(e: Env, from: Address, offer_id: i128) {
    |                            ^^^^ help: if this is intentional, prefix it with an
    ↪  underscore: `_from`
    |
    = note: `#[warn(unused_variables)]` on by default

warning: unused variable: `from`
   --> src/contract.rs:238:27
    |
238 |     fn close_loan(e: Env, from: Address, offer_id: i128) {
    |                           ^^^^ help: if this is intentional, prefix it with an
    ↪  underscore: `_from`
```