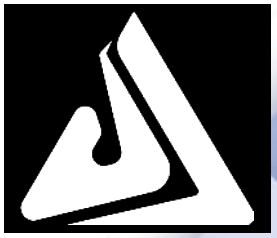
Veridise

Auditing Report

Hardening Blockchain Security with Formal Methods

FOR



OrbitCDP

Veridise Inc.

Dec. 26, 2024

► **Prepared For:**

Zenith Protocols Inc
https://orbitcdp.finance

► **Prepared By:**

Alberto Gonzalez
Evgeniy Shishkin
Victor Faltings

► **Contact Us:**

contact@veridise.com

► **Version History:**

Dec. 26, 2024 V1

Contents

1 Executive Summary

From Dec. 16, 2024 to Dec. 20, 2024, Zenith Protocols Inc engaged Veridise to conduct a security assessment of their OrbitCDP protocol. The OrbitCDP is a protocol in the Soroban platform, designed to allow users to borrow stablecoins pegged by fiat currencies using Stellar Lumens as collateral. Veridise conducted the assessment over 3 person-weeks, with 3 security analysts reviewing the project over 1 week on commit `311f35f`. The review strategy involved extensive manual code review performed by Veridise security analysts.

Project Summary. The OrbitCDP protocol leverages isolated liquidity pools from the Blend protocol to provide users with stablecoins pegged to fiat currencies. The protocol consists of four different modules that work together to facilitate the issuance and management of these stablecoins.

- **Treasury**. This module is responsible for minting and burning stablecoins. For a given currency, such as USD, an associated stablecoin (e.g., oUSD) is created. The treasury can freely mint and burn oUSD to control its total supply. When oUSD is minted, it is deposited into a Blend pool, allowing users to borrow it against cryptocurrency collateral, such as Stellar Lumens.
- **Bridge Oracle**. This module provides a reliable price feed for the various stablecoins. This price data is used by the Blend pools in all lending and borrowing operations to compute the positions' health factors.
- **Pegkeeper**. This module helps maintain the stablecoin pegs through Automated Market Maker (AMM) operations. It uses the Blend protocol's liquidity auction mechanism to purchase collateral at discounted prices from undesirable Blend pool positions. The acquired collateral is converted back into stablecoins through an AMM and used to pay back the debt to the Blend pool. To perform these operations, the Pegkeeper utilizes flashloans provided by the treasury.
- **Admin**. This module is responsible for managing the entire system. It calls privileged operations on the Treasury, Bridge Oracle, and Pegkeeper, such as adjusting the supply of stablecoins or updating various contracts. The aim is to have the admin controlled by a DAO contract.

Code Assessment. The OrbitCDP developers provided the source code of the contracts for review. The source code appears to be mostly original code written by the OrbitCDP developers. To assist Veridise analysts in comprehending the code, developers supplied detailed documentation outlining the project's intended business logic and design decisions. They also conducted a walk-through with the Veridise team, highlighting potential areas of concern and maintaining active communication throughout the code review process.

The source code included a test suite that the Veridise security analysts noted covered most of the workflow of the protocol, including the supply and withdrawal of stablecoins into Blend, the liquidation process, and oracle queries.

Summary of Issues Detected. The security assessment identified 11 issues, 3 of which were assessed as having a Medium level of severity by the Veridise team. Specifically, the analysts identified that the liquidation workflow in the Pegkeeper contract was not sharing a portion of the profits with the Orbit protocol, as outlined in the project's documentation (V-OBT-VUL-002). Additionally, the analysts identified the potential misuse of treasury flash loans for arbitrary purposes, rather than being restricted to their intended use for liquidations (V-OBT-VUL-003). The Veridise team also identified 2 low-severity issues. Notably, they found that it would have been possible to leave the Pegkeeper contract with an open debt position in the Blend protocol during the liquidation workflow (V-OBT-VUL-004). Additionally, the Veridise team highlighted the potential risk associated with transferring the administrator account in a single step (V-OBT-VUL-005). Moreover, the Veridise analysts identified 5 warnings and 1 informational finding.

Recommendations. After conducting the assessment of the protocol, the security analysts had a few suggestions to improve the OrbitCDP protocol security before deployment:

Soroswap Pool. Given the permissionless nature of Soroswap, which allows any user to create a trading pair, the analysts recommend caution regarding the absence of an AMM pool at the launch of the Orbit protocol. Users who acquire Orbit tokens, for instance through the Blend Pool, may create a pair with XLM on Soroswap at any price they choose.

Oracle TWAP Prices. Consider using Time-Weighted Average prices (TWAP) instead of spot prices when retrieving the actual data from the oracle. This could make the protocol more resistant to potential price manipulation attacks.

2 Project Dashboard

Table 2.1: Application Summary.

Name	Version	Type	Platform
OrbitCDP	`311f35f`	Rust	Stellar

Table 2.2: Engagement Summary.

Dates	Method	Consultants Engaged	Level of Effort
Dec. 16–Dec. 20, 2024	Manual	3	3 person-weeks

Table 2.3: Vulnerability Summary.

Name	Number	Acknowledged	Fixed
Critical-Severity Issues	0	0	0
High-Severity Issues	0	0	0
Medium-Severity Issues	3	2	1
Low-Severity Issues	2	2	1
Warning-Severity Issues	5	4	3
Informational-Severity Issues	1	1	1
TOTAL	11	9	6

Table 2.4: Category Breakdown.

Name	Number
Data Validation	5
Access Control	2
Usability Issue	2
Logic Error	1
Maintainability	1

3 Security Assessment Goals and Scope

3.1 Security Assessment Goals

The engagement was scoped to provide a security assessment of OrbitCDP's smart contracts. During the assessment, the security analysts aimed to answer questions such as:

- Are there any common Soroban implementation flaws, such as incorrect storage management, incorrect build order, or others?
- Are sensitive functions protected by appropriate access controls, and are the contracts' initializations implemented correctly to prevent unauthorized or unintended use?
- Does the flash loan implementation ensure that borrowed funds are returned within the same transaction or panics otherwise?
- Are there denial-of-service (DoS) attack vectors that could disrupt liquidation workflows or supply adjustments?
- Does the Bridge Oracle contract correctly forward the Blend Pool's requests to the configured oracle (e.g., Reflector) in compliance with the SEP-40 standard*?
- Does the protocol implementation reflect the higher-level business intent?

3.2 Security Assessment Methodology & Scope

Security Assessment Methodology and Scope. To address the questions above, the security assessment involved a thorough review by human experts.

The scope of this security assessment was limited to the below folders of the source code provided by the OrbitCDP developers. The Veridise analysts referenced the Blend protocol source code as necessary to validate its correct integration within the OrbitCDP codebase.

1. `admin/`
   a) `contract.rs`
   b) `storage.rs`
2. `bridge-oracle/`
   a) `contract.rs`
   b) `storage.rs`
3. `pegkeeper/`
   a) `contract.rs`
   b) `storage.rs`
   c) `helper.rs`
4. `treasury/`

* The SEP-40 proposal can be found on Stellar's ecosystem repository at https://github.com/stellar/stellar-protocol/blob/master/ecosystem/sep-0040.md

a) `contract.rs`
b) `storage.rs`

Methodology. Veridise security analysts inspected the provided tests, and read the OrbitCDP documentation. They then began a review of the code performing extensive manual review.

During the security assessment, the Veridise security analysts regularly asked the OrbitCDP developers questions about the code to understand the intended functionality.

3.3 Classification of Vulnerabilities

When Veridise security analysts discover a possible security vulnerability, they must estimate its severity by weighing its potential impact against the likelihood that a problem will arise.

The severity of a vulnerability is evaluated according to the Table 3.1.

Table 3.1: Severity Breakdown.

	Somewhat Bad	Bad	Very Bad	Protocol Breaking
Not Likely	Info	Warning	Low	Medium
Likely	Warning	Low	Medium	High
Very Likely	Low	Medium	High	Critical

The likelihood of a vulnerability is evaluated according to the Table 3.2.

Table 3.2: Likelihood Breakdown

Not Likely	A small set of users must make a specific mistake
Likely	Requires a complex series of steps by almost any user(s) - OR - Requires a small set of users to perform an action
Very Likely	Can be easily performed by almost anyone

The impact of a vulnerability is evaluated according to the Table 3.3:

Table 3.3: Impact Breakdown

Somewhat Bad	Inconveniences a small number of users and can be fixed by the user
Bad	Affects a large number of people and can be fixed by the user - OR - Affects a very small number of people and requires aid to fix
Very Bad	Affects a large number of people and requires aid to fix - OR - Disrupts the intended behavior of the protocol for a small group of users through no fault of their own
Protocol Breaking	Disrupts the intended behavior of the protocol for a large group of users through no fault of their own

4 Vulnerability Report

This section presents the vulnerabilities found during the security assessment. For each issue found, the type of the issue, its severity, location in the code base, and its current status (i.e., acknowledged, fixed, etc.) is specified. Table 4.1 summarizes the issues discovered:

Table 4.1: Summary of Discovered Vulnerabilities.

ID	Description	Severity	Status
V-OBT-VUL-001	Centralization Risk	Medium	Acknowledged
V-OBT-VUL-002	Pegkeeper doesn't share profits with protocol	Medium	Intended Behavior
V-OBT-VUL-003	Lack of Blend pool address validation . . .	Medium	Fixed
V-OBT-VUL-004	Pegkeeper can end with an open debt position	Low	Fixed
V-OBT-VUL-005	Admin transfer mistakes are irreversible	Low	Acknowledged
V-OBT-VUL-006	Sanity checks when adding a stablecoin	Warning	Fixed
V-OBT-VUL-007	OUSD token pegged to USDC instead of USD	Warning	Acknowledged
V-OBT-VUL-008	PegKeeper function name call is not validated	Warning	Intended Behavior
V-OBT-VUL-009	Missing slippage protection can make . . .	Warning	Fixed
V-OBT-VUL-010	Unbounded data is saved into instance storage	Warning	Fixed
V-OBT-VUL-011	Maintainability issues	Info	Fixed

4.1 Detailed Description of Issues

4.1.1 V-OBT-VUL-001: Centralization Risk

Severity	Medium	**Commit**	311f35f
Type	Access Control	**Status**	Acknowledged
File(s)	See description		
Location(s)	See description		
Confirmed Fix At	N/A		

Similar to many projects, OrbitCDP's contracts declare an administrator role that is given special permissions. In particular, this administrator is given the following abilities:

- Upgrade any of the contracts in the protocol.
- Control the blend pool interest rate by increasing or decreasing the OUSD supply.
- Add support to new stablecoins.
- Change the oracle used by blend pools.

Impact If a private key were stolen, a hacker would have access to sensitive functionality that could compromise the project. Most importantly, the administrator is given the ability to upgrade any of the contracts which will allow the minting of arbitrary amounts of supported stable tokens disrupting all the Orbit ecosystem around them.

Recommendation As these are all particularly sensitive operations, we would encourage the developers to utilize decentralized governance or multi-sig contract as opposed to a single account, which introduces a single point of failure.

Developer Response The administrator will be managed by a DAO contract. Specifically, the project plans to use the Soroban Governor contract.

4.1.2 V-OBT-VUL-002: Pegkeeper doesn't share profits with protocol

Severity	Medium	**Commit**	311f35f
Type	Logic Error	**Status**	Intended Behavior
File(s)	`pegkeeper/src/contract.rs`		
Location(s)	fl_receive()		
Confirmed Fix At	N/A		

The "Key Features" section of the protocol description here states that profits from liquidated debt positions are shared between the protocol and the user-liquidator. However, the code doesn't reflect that and sends all the profits to the user.

```
fn fl_receive(e: Env, token: Address, amount: i128, blend_pool: Address,
      auction: Address, collateral_token: Address,
      lot_amount: i128, liq_amount: i128, amm: Address, fee_taker: Address) {
  // ... skipped ...
  if balance_before > balance_after {
     panic_with_error!(&e, PegkeeperError::NotProfitable);
  }
  let profit = balance_after - balance_before;
  token_client.transfer(&e.current_contract_address(), &**fee_taker**, &profit);\
  // ... skipped ...
}
```

Snippet 4.1: Snippet from `fl_receive()`

Impact The protocol becomes less profitable for the owners than it could be.

Recommendation If this is unintended behavior, implement the rewarding logic correctly. Otherwise, the documentation has to be fixed.

Developer Response The developers acknowledged that no sharing profits with the Orbit protocol is the intended behavior and the documentation will be updated accordingly.

4.1.3 V-OBT-VUL-003: Lack of Blend pool address validation allows users to take arbitrary flash-loans

Severity	Medium	**Commit**	311f35f
Type	Data Validation	**Status**	Fixed
File(s)	`treasury/src/contract.rs`		
Location(s)	keep_peg()		
Confirmed Fix At	`https://github.com/orbit-cdp/orbit-contracts/pull/9`, 6fbbe30		

Users can call the `keep_peg()` function of the treasury to liquidate debt positions through the PegKeeper when the collateral token loses too much value. Before calling PegKeeper, the Treasury's `keep_peg()` function mints the requested amount of stablecoins for PegKeeper and transfers them to the specified `blend_pool` address. After liquidation, this amount must be transferred back from PegKeeper's balance.

However, since the `blend_pool` address is not verified in any way, a malicious user could provide their own contract address that implements a Blend-compatible interface and benefit from a flash loan without paying any fees to the protocol. In this case, the only requirement for the malicious user is to provide a minimum amount of tokens for exchange on Soroswap, as Soroswap does not allow zero-amount swaps.

Although neither the `amm` nor the `collateral_token` values are validated in PegKeeper, a malicious user could provide their own Soroswap pair address with an exchange rate and artificial token, minimizing their spending.

Impact Users are able to take nearly free flash loans in arbitrary amounts. This allows them to use treasury flash loans for operations beyond their original intended purpose, which is to perform liquidations. This capability could have an impact beyond the Orbit protocol, as these flash loans provide a foundation for other attacks that require substantial liquidity, such as price manipulation attacks.

Recommendation It is recommended to implement the check in the Treasury's `keep_peg()` function that would ensure that the provided Blend Pool address corresponds to the correct stable coin token.

Developer Response The developers implemented the suggested fix.

4.1.4 V-OBT-VUL-004: Pegkeeper can end with an open debt position

Severity	Low	**Commit**	311f35f
Type	Data Validation	**Status**	Fixed
File(s)	`pegkeeper/src/contract.rs`		
Location(s)	fl_receive()		
Confirmed Fix At	`https://github.com/orbit-cdp/orbit-contracts/pull/7`, 088b84d		

In the pegkeeper's `fl_receive(...)` function, the pegkeeper is tasked with liquidating a position and making profit from underpriced collateral. This process is performed in several steps:

1. Receive an `amount` of stablecoins (e.g. oUSD) minted by the treasury as a flash loan.
2. Liquidate a position in the Blend pool and receive collateral tokens at a discount.
3. Swap these collateral tokens into stablecoins (e.g. oUSD) through an AMM.
4. Transfer the calculated profits to `fee_taker`.
5. Pay back the flash loan.

Inspecting the code in more detail, the second step of this process is itself a combination of 3 requests submitted to the Blend pool:

1. Fill a given liquidation position.
2. Repay `amount` debt of this position (in stablecoins).
3. Withdraw `lot_amount` collateral from this position (in collateral tokens).

The one restriction imposed by this set of requests is that the final state of the Blend user's position (in this case the position of the pegkeeper) is healthy. However, it is not required to repay the debt in full. This could create a situation in which the pegkeeper is left with outstanding debt, if the caller of `keep_peg` supplies an `amount` and `lot_amount` that are less than the actual debt and collateral balance of the pegkeeper position after the liquidation.

```
let fill_requests = vec![
    e,
    Request {
        request_type: 6 as u32, // liquidationAuction
        address: auction_creator.clone(),
        amount: liq_amount.clone(), // No constraints w.r.t. other requests
    },
    Request {
        request_type: 5 as u32, // Repay
        address: token_a.clone(),
        amount: token_a_bid_amount, // User-provided
    },
    Request {
        request_type: 3 as u32, // Withdraw
        address: token_b.clone(),
        amount: token_b_lot_amount, // User-provided
    },
];
```

Snippet 4.2: Snippet from `pegkeeper/helper::liquidate()`

Impact In the end the pegkeeper or treasury will not lose any assets from this. However, the caller of `keep_peg` would lose out on potential profits since they are not withdrawing the full collateral balance of the auctioned position. Additionally, the pegkeeper could accrue debt which might have undesirable impacts on future transactions.

Recommendation The `fl_receive(...)` function should check that the pegkeeper does not have any liabilities after liquidating the auctioned position. Additionally, it should be checked that all the acquired collateral was withdrawn in order to maximize profits.

Developer Response The developers implemented the suggested fix.

4.1.5 V-OBT-VUL-005: Admin transfer mistakes are irreversible

Severity	Low	**Commit**	311f35f
Type	Access Control	**Status**	Acknowledged
File(s)	`admin/src/contract.rs`		
Location(s)	set_admin()		
Confirmed Fix At	N/A		

The `set_admin()` function directly transfers admin rights without any confirmation or acceptance from the proposed new admin. This means that if the current admin mistakenly sets the wrong address, the transfer is immediate and irreversible.

Impact The impact of this issue could be significant, as an error in transferring admin rights will lead to a loss of control over the contract's administrative functions.

Recommendation To address this issue, it is recommended to implement a two-step transfer process for admin rights. This process should include an initial request to transfer admin rights, followed by a requirement for the proposed new admin to accept the transfer.

Developer Response The developers plan to use a DAO contract as the administrator account which reduces the risk of making a mistake. To avoid the necessity to pass a proposal for letting the DAO become admin of the protocol, the developers will maintain the current design.

4.1.6 V-OBT-VUL-006: Sanity checks when adding a stablecoin

Severity	Warning	**Commit**	311f35f
Type	Data Validation	**Status**	Fixed
File(s)	`admin/src/contract.rs`		
Location(s)	new_stablecoin()		
Confirmed Fix At	`https://github.com/orbit-cdp/orbit-contracts/pull/9`, 6fbbe30		

The `new_stablecoin()` function in the `admin` contract is responsible for adding support for new stablecoins to the OrbitCDP protocol. This function takes several parameters, including `token` (the address of the new stablecoin) and `blend_pool` (the address of the blend pool where the initial stablecoin supply will be allocated).

Currently, the function lacks necessary validations:

1. **Duplicate Token Check**: There is no mechanism to ensure the `token` has not already been added, which may lead to overwriting existing entries.
2. **Blend Pool Validation**: The `blend_pool` is not verified to be a legitimate pool using the blend factory's `is_pool()`function, increasing the risk of adding contracts outside the Blend protocol.

Impact The lack of validations in the `new_stablecoin()` function can cause the following issues:

1. Overwriting existing token configurations.
2. Associating the protocol with pools not deployed by the Blend protocol.

Recommendation To address these issues, it is recommended to implement checks within the `new_stablecoin()` function to ensure that the `token` has not been previously added. Additionally, incorporate a validation mechanism to confirm the legitimacy of the `blend_pool`, by invoking the blend's factory function `is_pool()` to ascertain its validity.

Developer Response The developers implemented the suggested fix.

4.1.7 V-OBT-VUL-007: OUSD token pegged to USDC instead of USD

Severity	Warning	**Commit**	311f35f
Type	Usability Issue	**Status**	Acknowledged
File(s)	`bridge_oracle/src/contract.rs`		
Location(s)	lastprice()		
Confirmed Fix At	N/A		

The `bridge-oracle` contract is designed to be an intermediary between blend pools and the price oracle network Reflector. In this way, the `lastprice()` function is responsible for fetching the latest price data for a given asset when asked by the blend pool and forwarding the request to Reflector.

In the current implementation, the `lastprice()` function retrieves the conversion target asset using `storage::get_bridge_asset()`. This setup implies that the OUSD token is configured to convert to USDC rather than USD. As a result, OUSD will be pegged to USDC, not directly to USD. This behavior might not be immediately apparent to users, leading to potential misunderstandings about the value of OUSD.

Impact The primary impact of this configuration is that users might assume OUSD is pegged directly to USD, while it is actually pegged to USDC. This could lead to discrepancies in expected versus actual value, especially if there are fluctuations in the USDC/USD exchange rate.

Recommendation To address this issue, it is recommended to clearly document this behavior in the contract's documentation and user-facing materials. Users should be informed that OUSD is pegged to USDC, not USD, to set accurate expectations. Additionally, consider evaluating whether a direct USD peg is feasible or more appropriate for the intended use case of OUSD.

Developer Response The developers acknowledged the issue and will update their documentation highlighting what oracle is used and what assets their orbit stable-coins are pegged to.

4.1.8 V-OBT-VUL-008: PegKeeper function name call is not validated

Severity	Warning	**Commit**	311f35f
Type	Data Validation	**Status**	Intended Behavior
File(s)	`treasury/src/contract.rs`		
Location(s)	keep_peg()		
Confirmed Fix At	N/A		

The Treasury contract has a keep_peg() function that can be called by anyone. This function takes two arguments: `name` and `args`.

The `name` parameter represents the function name of the method that will be executed in the PegKeeper contract. This value is not validated, so any method in the PegKeeper can be called.

Currently, the only method available to users is `fl_receive()`, so there is little point in validating it. However, if the PegKeeper contract is extended in the future, not having proper checks in place in this part of the protocol could pose a security risk.

```
fn keep_peg(e: Env, name: Symbol, args: Vec<Val>) {
  storage::extend_instance(&e);
  let token = Address::try_from_val(&e, &args.get(0).unwrap()).unwrap();
  let amount = i128::try_from_val(&e, &args.get(1).unwrap()).unwrap();
  let pegkeeper: Address = storage::get_pegkeeper(&e);
  // ... skipped ...
  e.invoke_contract::<Val>(&pegkeeper, &name, args.clone());
  // ... skipped ...
}
```

Snippet 4.3: Snippet from `keep_peg()`

Impact There is currently no security concern. However, if the PegKeeper contract is ever extended, the lack of validation in this function could potentially pose a security risk.

Recommendation It is recommended to implement a simple check on the `name` argument to rule out this potential risk.

Developer Response The developers acknowledged that this is the intended behavior of the current design.

4.1.9 V-OBT-VUL-009: Missing slippage protection can make liquidator walkout without profit

Severity	Warning	**Commit**	311f35f
Type	Data Validation	**Status**	Fixed
File(s)	`pegkeeper/src/contract.rs`		
Location(s)	fl_receive()		
Confirmed Fix At	`https://github.com/orbit-cdp/orbit-contracts/pull/7/`, 0af320e		

The `fl_receive()` function in the `PegkeeperContract` is responsible for processing flashloans initiated by the `treasury`contract to liquidate unhealthy accounts in the Blend protocol, swap the collateral, and distribute profits to the liquidator. While the function ensures that the balance after operations is greater than before (indicating a profit), it lacks a mechanism to guarantee that the profit meets a minimum threshold specified by the liquidator.

Impact Without a minimum profit threshold, liquidators may execute transactions that are economically unviable, leading to wasted gas fees or losses due to insignificant gains.

Recommendation To address this issue, it is recommended to implement a check that ensures the profit exceeds minimum threshold defined by the liquidator.

Developer Response The developers implemented the suggested fix.

4.1.10 V-OBT-VUL-010: Unbounded data is saved into instance storage

Severity	Warning	**Commit**	311f35f
Type	Usability Issue	**Status**	Fixed
File(s)	`bridge-oracle/src/storage.rs`, `treasury/src/storage.rs`		
Location(s)	set_bridge_asset(), set_blend_pool()		
Confirmed Fix At	`https://github.com/orbit-cdp/orbit-contracts/pull/8`, e72d5e2		

The `treasury/src/storage.rs` and `bridge-oracle/src/storage.rs` files contain the functions `set_blend_pool()` and `set_bridge_asset()` which store the keys `BLENDPOOL(token_address.clone())` and `BRIDGE(asset.clone())` in the instance storage of the contracts.

The issue arises because these keys are unbounded in nature; any number of assets can be added to the configuration of these contracts. Instance storage is not recommended for storing unbounded data due to its limited capacity. This design can lead to increased costs for users since all instance storage is loaded, even if some entries are not used during the current execution. This inefficiency can degrade performance and increase operational costs.

Impact The use of instance storage for unbounded data can result in performance bottlenecks and elevated costs for users.

Recommendation To address this issue, consider using persistent storage to save the aforementioned keys.

Developer Response The developers implemented the suggested fix.

4.1.11 V-OBT-VUL-011: Maintainability issues

Severity	Info	**Commit**	311f35f
Type	Maintainability	**Status**	Fixed
File(s)	See description		
Location(s)	See description		
Confirmed Fix At	`https://github.com/orbit-cdp/orbit-contracts/pull/6`, c6bfbf5		

The analysts identified several areas of the code that require improvement in order to prevent possible maintainability issues in the future. These include:

- The use of literals in place of named constants.
  In several places in the code, developers have used numeric literals rather than named constant values. The following files are affected:
  - `pegkeeper/src/contract.rs`
  - `pegkeeper/src/helper.rs`
  - `treasury/src/contract.rs`
- Variable shadowing.
  In the PegKeeper contract, there is a function called `fl_receive`, and the argument `lot_amount` is shadowed by a local variable with the same name.
- Imprecise or incorrect comments.
  - The comment on line 12 of `admin/src/contract.rs` says : "Initializes the bridge oracle" but it should actually be "Initializes the AdminContract".
  - The comment on line 10 of `pegkeeper/src/contract.rs` says: "Initialize the treasury", but it should say "Initializes the PegKeeper contract". Also, the arguments for the initialization function are not clearly described. There is no specific argument named `maximum_duration`. Instead, the argument for the router address should be specified.
  - The comment on line 24 of `pegkeeper/src/helper.rs` says "Withdrawal", however, the numeric literal "3" corresponds to "Collateral withdrawal", while the usual withdrawal operation is denoted by a different number.
- Use of deprecated interface.
  In the PegKeeper contract, the `fl_receive` function makes use of the `token::Client` type on lines 53 and 54. The `token::TokenClient` type should be used instead, according to the source code of the library (`https://docs.rs/soroban-sdk/latest/src/soroban_sdk/token.rs.html#26`).
- Confusing name of `admin` in the `pegkeeper` contract.
  In the `pegkeeper` contract the administrator account is assigned to the `admin` variable. However, the administrator of this contract is expected to be the `treasury` contract and not the `admin` contract. In this way it is advisable to change `admin` for `treasury` to avoid any confusion.
- Unchecked return value in the `pegkeeper/helper` contract.
  In the `swap` function of the `pegkeeper/helper` contract, the function calls the `swap_exact_tokens_for_tokens` function from the Soroswap interface. The return value of this function is never assigned, which may give the impression that the function does not return anything. It would be better to check this return value or explicitly ignore it to help future maintainability.

Impact There are currently no security implications. However, the aforementioned issues may cause confusion for developers and lead to maintainability problems.

Recommendation It is recommended to fix all the listed issues.

Developer Response The developers implemented the suggested fixes.

Glossary

flashloan A loan which must be repaid in the same transaction, typically offered at a much more affordable rate than traditional loans. 1

Soroban Soroban is the smart contracts platform on the Stellar network. See `https://stellar.org/soroban` to learn more. 1

Stellar Lumen Native token of the Stellar Network. See `https://stellar.org/learn/lumens` to learn more . 1