Slender Security Assessment Report

July 2024

Prepared for Slender

Table of content

Project Summary

Project Scope

The original project files are:

Repository	Files	Commits	Compiler	Platform
eq-lab/slender/	All files in the repo	(original) 93d1648		Stellar
eq-lab/slender/	All files in the repo	(audit-fixes) 993fea5		Stellar

Protocol Overview

Slender is the first non-custodial lending/borrowing DeFi protocol on Stellar's Soroban network. It uses a pool-based strategy that aggregates each user's supplied assets. Currently, the protocol supports only three markets (XRP, XLM, and USDC) but ultimately plans to expand in order to allow users to lend and borrow any asset that is supported on Soroban (including real-world assets).

Lenders provide liquidity to a market in exchange for an s-token (i.e., essentially Slender's LP-Token). These tokens accrue interest and reflect this accrual in their "price". Users are able to borrow assets from the protocol via an over-collateralized loan which issues them a dToken (debt token). Each borrowing market has a floating interest rate, determined by the utilization of that market's assets. Utilization is capped (default value is 90% cap) to always keep a reserve for user withdrawals. Users cannot be both lenders and borrowers of the same asset.

In addition, users can take flash loans from a market. When a flash loan is concluded, the users have a choice between paying back the loan + fee or borrowing the funds in which case no fee is charged.

For price information Slender's plans to use SEP-40 compatible third-party oracles (e.g., reflector) and apply TWAP to the price data points in order to mitigate operator error, market volatility, and manipulation risks.

Project Goals

Assess the security of the protocol via manual audit.

Findings Summary

The table below summarizes the findings of the review, including type and severity details.

Severity	Discovered	Acknowledged	Code Fixed
Critical	5	5	5
High	3	3	3
Medium	7	7	6
Low	1	1	1
Informational	4	3	3
Total	20	19	18

Detailed Findings

C-1 Users can combine borrow + withdraw to open positions with arbitrary (positive) NPV		
Severity: **Critical**	Impact: **High**	Likelihood: **High**
Category: Logic, Economics		Files: borrow.rs withdraw.rs finalize_transfer.rs

Description

In order to ensure the solvency of an over-collateralized borrowing & lending protocol, there is usually a minimal loan-to-value ratio that is required in order to open a position. In Slender's case this is expressed via the initial_health configuration parameter which is checked in line #105 of the `do_borrow` function:

```
    require_gte_initial_health(env, &account_data, amount_in_base)?;
```

However, we note that the withdraw function only checks that users have *positive* NPV (line #114 for fungible assets and #164 for real-world assets) but does *not* check that the NPV is greater or equal to initial health:

```
                require_good_position(env, &account_data);
```

Similarly, the only check on users performing a transfer of s-tokens (in L#72 of finalize_transfer.rs) is:

```
        require_good_position(env, &from_account_data);
```

That is not a priori unreasonable, but effectively makes it possible to bypass said restriction which is imposed in the case of borrowing.

Exploit

There are (at least) two possible attack vectors:

1. An attacker can borrow and then withdraw in order to open “bad positions” (i.e., positions with less than the required initial health), causing the protocol to accumulate bad debt.
2. An attacker can open a position with precisely the minimal required initial health and then withdraw and self-liquidate to extract money from the protocol.

Recommendation

Either have an intermediate “under-water” stage between healthy position and liquidation (i.e., when 0 = liquidation_threshold < position_health < initial_health) where users are only allowed to deposit and repay but withdraw is disabled or require the user to cover a proportional part of their total debt when withdrawing their collateral.

Customer Response

Acknowledged and will fix.

Fix Review

The issue is successfully resolved in the commit 993fea5 by adding the missing check (i.e., see PR #128).

C-2 Transfer- and Burn-on-zero can cause liquidation to revert

Severity: **Critical**	Impact: **High**	Likelihood: **High**
Category: Logic, Economics		Files: [liquidate.rs]()

Description

The functions

```
   fn transfer_on_liquidation(e: Env, from: Address, to: Address, amount: i128) {
       verify_caller_is_pool(&e);
       require_positive_amount(amount);

       do_transfer(&e, from, to, amount, false);
   }
```

And the functions (called in `do_transfer`):

```
pub fn receive_balance(e: &Env, addr: Address, amount: i128)
pub fn spend_balance(e: &Env, addr: Address, amount: i128)
```

all panic if the amount parameter is non-positive. Thus, because of the line

```
              s_token.transfer_on_liquidation(who, liquidator, &liq_lp_amount);
```

Liquidation would revert if `liq_lp_amount` turns out to be non-positive. The latter is computed as:

```
           let liq_lp_amount = FixedI128::from_inner(collat.coeff.unwrap())
               .recip_mul_int(liq_comp_amount)
               .ok_or(Error::LiquidateMathError)?;
```

and

```
       // the same for token-based RWA
       let liq_comp_amount = calc_liq_amount(
           price_provider,
           &collat,
           hundred_percent,
           discount_percent,
           liq_bonus_percent,
           safe_collat_percent,
           initial_health_percent,
           total_collat_disc_after_in_base,
           total_debt_after_in_base,
       )?;
```

However, examining the code it is not hard to see that `liq_comp_amount` can be zero:

```


    let liq_comp_amount = price_provider.convert_from_base(&collat.asset,
safe_collat_in_base)?;

    let liq_comp_amount = safe_discount_percent
        .recip_mul_int(liq_comp_amount)
        .ok_or(Error::LiquidateMathError)?;

    Ok(if liq_comp_amount.is_negative() {
        collat.comp_balance
    } else {
        collat.comp_balance.min(liq_comp_amount)
    })
```

A similar issue occurs for burn as well in L#176:

```
                s_token.burn(who, &liq_lp_amount, &liq_comp_amount, liquidator);
                add_stoken_underlying_balance(env, &s_token.address, amount_to_sub)?;
```

Recommendation

In liquidations, allow transfer and burn of zero value to prevent reverts.

Customer Response

Acknowledged and will fix.

Fix Review

Fixed in commit 993fea5 by adding checks before transferring the underlying asset from the liquidator to the s-token contract address (L#295-297):

```
            if debt_comp_to_transfer > 0 {
                underlying_asset.transfer(liquidator, s_token_address,
&debt_comp_to_transfer);
            }
```

and before burning the debt token (L#299-301)

```
            if debt_lp_to_burn > 0 {
                debt_token.burn(who, &debt_lp_to_burn);
            }
```

And the s-token (L#195-197):

```
            if liq_lp_amount > 0 && liquidator_part_underlying > 0 {
                s_token.burn(who, &liq_lp_amount, &liquidator_part_underlying,
liquidator);
            }
```

C-3 Stellar's resource limit can block liquidations		
Severity: **Critical**	Impact: **High**	Likelihood: **High**
Category: Blockchain		Files: [liquidate.rs]()

Description

Due to Stellar's limit on I/O operations the liquidation function will revert as soon as the number of reserves the user is using as collateral or borrowing is sufficiently large (e.g., more than ~5) or even with 3 assets if the cost of oracle prices query (which can be proportional to the amount of TWAP records requested) is sufficiently large.

Impact

In such cases, users might be able to borrow assets from the protocol while being immune to liquidations, effectively creating bad debt and threatening the economic stability of the protocol.

Recommendation

Carefully check the cost of external calls like prices or underlying_asset.transfer in terms of I/O operations and adjust parameters accordingly. Implement logic that prevents a user from depositing or borrowing from too many different reserves.

Customer Response

Acknowledged and fixed. We added settings to limit the number of active reserves (see L#12 of pool_config.rs) per user. Currently, it is set to 3, but when the Stellar blockchain increases their I/O limits, it will be reconsidered. Furthermore, we have updated the Oracle mock contract and reproduced the approach reflector's price oracle (the oracle used for launch) implements when returning prices. After testing the most CPU/Memory intensive operations (liquidation with 3 assets: 2 deposits + 1 debt, 1 deposit + 2 debts) we are going to reduce the number of latest price values for each asset to 3 for now (see L#12 of price_feed.rs).

Fix Review

The problem is resolved in commit 993fea5 via the changes outlined above.

C-4 Liquidating small positions is not incentivized

Severity: **Critical**	Impact: **High**	Likelihood: **High**
Category: Logic		Files: liquidate.rs

Description

If a user's borrowing balance exceeds their total collateral value (borrowing capacity) due to the value of collateral falling, or borrowed assets increasing in value, the liquidation mechanism implemented in Slender allows the liquidator to buy the borrower's collateral (at a slightly better than market price). Hence, the liquidation bonus is essentially implemented here as a percentage of a bad position's total collateral. However, we note that there is no restriction on opening positions with small values of collateral/debt as long as the user's NPV is positive, so a bad actor can initiate a sybil attack on the protocol, accruing bad debt that can lead to protocol insolvency.

Exploit

An attacker opens multiple positions (via a sybil attack) with small collateral and debt values and each with a small total NPV and waits for them to deep into negative NPV on purpose. No liquidator actually has the incentive to liquidate each of the individual positions because the cost of submitting a liquidating transaction is greater than the potential gain. Thus the attacker manages to deliberately create bad debt for the protocol.

Recommendation

Disallow repayments/withdrawals which leave only a small debt/collateral (the precise definition of "small" here should probably be a configuration parameter determined via off-chain simulation).

Customer response

Acknowledged and will fix.

Fix Review

The issue is resolved in the latest commit by adding the `pool_config.min_collat_amount` and `pool_config.min_debt_amount` parameters and the check

```
                require_min_position_amounts(env, &account_data, &pool_config)?;
```

in line #158 of repay.rs, line #150 of borrow.rs, line #84 of finalize_transfer.rs, line #50 of set_as_collateral.rs, and line #187 of withdraw.rs.

C-5 Incorrect rounding enables an attacker to drain funds from the protocol

Severity: **Critical**	Impact: **High**	Likelihood: **High**
Category: Arithmetic		Files: withdraw.rs

Description

It is essential that rounding in DeFi would always favor the protocol. However, examining the code of withdraw.rs which handles the amount of underlying asset token to supply to the user and the corresponding amount of s-tokens to burn for fungible reserve types, we note that the key quantity `s_token_to_burn` is *rounded down instead of up*.

```
        let (underlying_to_withdraw, s_token_to_burn) =
        if amount >= underlying_balance {
            (underlying_balance, collat_balance)
        } else {
            let s_token_to_burn = collat_coeff
                .recip_mul_int(amount)
                .ok_or(Error::MathOverflowError)?;
            (amount, s_token_to_burn)
        };
```

Exploit

In our case, the core idea behind the attack is elementary:
assume that `collat_coeff>1`, say for instance `collat_coeff = 2`. Alice (the attacker) deposits `amount=4` A-tokens, and receives `amount_to_mint = 2` s-tokens in return. Alice then proceeds to request to withdraw 3 A-tokens from the reserve. Since `s_token_to_burn = 3/2 = 1`, it can do it again, receiving a total of `6 = 3 + 3` A-tokens in the expanse of the protocol and other users.

Recommendation

- Fix the rounding error (i.e., in line #62 of withdraw.rs - replace `.recip_mul_int` with `.recip_mul_int_ceil`).

Customer response

Acknowledged and will fix.

Fix review

Resolved in the latest commit. The entire computation above has now been refactored into the function

```
pub fn get_lp_amount(
    env: &Env,
    reserve: &ReserveData,
    pool_config: &PoolConfig,
    s_token_supply: i128,
    s_token_underlying_balance: i128,
    debt_token_supply: i128,
    amount: i128,
    round_ceil: bool,
) -> Result<i128, Error>
```

which is called in withdraw.rs with the `round_ceil`l boolean flag set to true:

```
            } else {
                let s_token_to_burn = get_lp_amount(
                    env,
                    &reserve,
                    &pool_config,
                    s_token_supply,
                    stoken_underlying_balance,
                    debt_token_supply,
                    amount,
                    true,
                )?;

                (amount, s_token_to_burn)
```

H-1 Liquidator can seize a bad position's collateral without repaying any of its debt		
Severity: **High**	Impact: **High**	Likelihood: **Medium**
Category: Logic, Economics		Files: liquidate.rs

Description

If a user's NPV is sufficiently negative, it is possible for the liquidation bonus to reach 100%:

```
let zero_percent = FixedI128::from_inner(0);
    let initial_health_percent =
FixedI128::from_percentage(read_initial_health(env)?).unwrap();
    let hundred_percent = FixedI128::from_percentage(PERCENTAGE_FACTOR).unwrap();
    let npv_percent = FixedI128::from_rational(account_data.npv,
total_collat_disc_after_in_base)
        .ok_or(Error::LiquidateMathError)?;
    let liq_bonus_percent = npv_percent.min(zero_percent).abs().min(hundred_percent);
```

Which would cause `debt_in_base` (and therefore `total_debt_to_cover_in_base`) to be zero:

```
    let total_debt_liq_bonus_percent = hundred_percent
        .checked_sub(liq_bonus_percent)
        .ok_or(Error::LiquidateMathError)?;
// . . .
        let debt_comp_amount = total_debt_liq_bonus_percent
            .mul_int(liq_comp_amount)
            .ok_or(Error::LiquidateMathError)?;

        let debt_in_base = price_provider.convert_to_base(&collat.asset,
debt_comp_amount)?;
```

Recommendation

We suggest fixing the mathematical/economical logic and the formula to account for this scenario.

Customer response

Acknowledged and will fix.

Fix Review

The latest commit [993fea5] fixes the issue.

H-2 Lenders can be immediately liquidated once the protocol is unpaused

Severity: **High**	Impact: **High**	Likelihood: **Medium**
Category: Logic		Files: liquidate.rs

Description

Slender contains the option to pause the protocol (by a privileged user rule) which correctly stops all protocol functions such as borrowing, depositing, liquidations, etc. However, once the protocol has been unpaused there is no "grace period" granted to lenders - if external price fluctuations have caused their position to fall under water they can be liquidated immediately without being given any chance to save their positions, which is unfair towards the users of the protocol.

Recommendation

Allow for AAVE-style "grace period[1]" when returning from a pause state to normal operation (see e.g., PriceOracleSentinel for a reference implementation of this idea).

Customer response

Acknowledged. Will Fix.

Fix Review

Fixed in the latest commit 993fea5.

[1] Note: during such "grace period" user operations such as deposits + repayments + flash loans should be allowed, but not withdrawals or token transfers.

H-3 TWAP price calculation can be incorrect

Severity: **High**	Impact: **Medium**	Likelihood: **High**
Category: Arithmetic Error, SEP-40 Oracle, Logic		Files: price_provider.rs

Description

As often recommended, Slender prices assets using the time-weighted average price (TWAP) of multiple oracle price data points in order to reduce the risk of high asset volatility or malicious price spoofing. However, the function twap which implements the computation incorrectly assumes that the PriceData vector

```
/// Price data for an asset at a specific timestamp
#[contracttype]
pub struct PriceData {
    price: i128,
    timestamp: u64
}
```

reported by the SEP-40 function prices

```
/// Get last N price records
fn prices(
    env: soroban_sdk::Env,
    asset: Asset,
    records: u32
) -> Option<Vec<PriceData>>;
```

is always sorted in *descending* order with respect to the timestamp:

```
          let price_curr = prices.get_unchecked(0); //@audit we implicitly assume
this is the most current price.
```

```
            let price_prev = prices.get_unchecked(i - 1);
            let price_curr = prices.get_unchecked(i); //@audit we implicitly assume
here prices are sorted in descending order with respect to timestamp
```

However, while this may be true for Reflector, it is not currently part of the trait defined by the SEP and would lead to an arithmetic error in general.

Exploit Scenario

Incorrect TWAP computations leads to systematically incorrect pricing of collateral and debt which would allow savvy arbitrage traders to drain the protocol.

Recommendation

Check to ensure that the prices received from the oracle are indeed sorted according to their timestamp in descending order.

Customer response

Acknowledged. Will fix.

Fix Review

This issue is resolved in commit 993fea5.

M-1 The formula for NPV in the technical specification is unclear		
Severity: **Medium**	Impact: **Low**	Likelihood: **High**
Category: Documentation		Files:

Description

In Slender's technical specification document, the formula for computing the net position value (NPV) is stated[2] as:

$$NPV_{specification} = \sum_i Collateral_i \cdot Discount_i \cdot Price_i - \sum_j Debt_j \cdot Price_j$$

However, examining the code in account_positions.rs, we see that NPV is actually computed as:

$$NPV_{code} = \sum_i CCollateral_{i,t} \cdot Discount_i \cdot Price_i - \sum_j CDebt_{j,t} \cdot Price_j$$

Where the *compound balance* and *debt* are the product of the collateral (respectively, debt) coefficient with the balance (resp. debt):

$$CCollateral_{i,t} = CC_t^i \cdot Collateral_i$$

$$CDebt_{j,t} = DC_t^j \cdot Debt_i$$

where collateral coefficient is computed via the formula

$$CC_t^i = \frac{total\_debt_i \cdot LAR_t^i + Balance_i}{sToken_i\ supply}$$

and the debt coefficient is another name for the borrower's accrued rate

$$DC_t^j = BAR_t^j$$

Impact

[2] Note: it is mentioned in the document that "NPV is also reduced when the interest rate is accrued: the debt term under the second sum is the *real* debt, *just like the collateral term under the first sum term.*", however the meaning of "real" in this context is not made clear.

Traders relying on the formula published by Slender could be liquidated despite the fact that by the stated formula their positions are supposed to be healthy.

Recommendation

Correct the discrepancy between the code and the published documentation by changing the notation to explicitly include the collateral/debt coefficients, expanding the subsequent explanatory paragraph, and perhaps including some numerical examples to illustrate the computation.

Customer response

Acknowledged, will change the documentation to reflect this.

Fix Review

Fixed in the latest version of the technical specification.

certora

M-2 Precision loss issues: division-before-multiplication		
Severity: **Medium**	Impact: **Medium**	Likelihood: **Medium**
Category: Arithmetic		Files: account_position.rs deposit.rs liquidate.rs withdraw.rs price_provider.rs

Description

The following computations are instances of division-before-multiplication which lead to a loss of accuracy -

(L#188-#190 of account_position.rs):

```
        let compounded_balance = collat_coeff
            .mul_int(who_collat)
            .ok_or(Error::CalcAccountDataMathError)?;
```

(L#94-#96 of deposit.rs):

```
    let amount_to_mint = collat_coeff
        .recip_mul_int(amount)
        .ok_or(Error::MathOverflowError)?;
```

(L#137-#139 of liquidate.rs):

```
            let liq_lp_amount = FixedI128::from_inner(collat.coeff.unwrap())
                .recip_mul_int(liq_comp_amount)
                .ok_or(Error::LiquidateMathError)?;
```

(L#54-#56 of withdraw.rs):

```
            let underlying_balance = collat_coeff
                .mul_int(collat_balance)
                .ok_or(Error::MathOverflowError)?;
```

(L#61-#63 of withdraw.rs):

```
               let s_token_to_burn = collat_coeff
                    .recip_mul_int(amount)
                    .ok_or(Error::MathOverflowError)?;
```

(L#41-#45 of price_provider.rs):

```
       median_twap_price
           .mul_int(amount)
           .and_then(|a| FixedI128::from_rational(a,
10i128.pow(config.asset_decimals)))
           .and_then(|a| a.to_precision(self.base_asset.decimals))
           .ok_or(Error::InvalidAssetPrice)
```

(L#56-#60 of price_provider.rs):

```
       median_twap_price
           .recip_mul_int(amount)
           .and_then(|a| FixedI128::from_rational(a,
10i128.pow(self.base_asset.decimals)))
           .and_then(|a| a.to_precision(config.asset_decimals))
           .ok_or(Error::InvalidAssetPrice)
```

Recommendation

Add arithmetic functions to ensure that division occurs after multiplication in this computation. For example we could replace L#188-#190 of account_position.rs with a function

```
/// Returns amount * collateral coefficient without losing precision
pub fn get_compounded_balance(
   env: &Env,
   reserve: &ReserveData,
   s_token_supply: i128,
   s_token_underlying_balance: i128,
   debt_token_supply: i128,
   amount: i128,
) -> Result<FixedI128, Error>
```

Which computes the compounded balance as (while being mindful of the potential for overflow of course!):

```
[(s_token_underlying_balance + lender_ar * debt_token_supply)*amount]/s_token_supply
```

Similarly for the other computations.

Customer Response

Acknowledged and will fix.

Fix Review

The issue is resolved in the recent commit 993fea5. As recommended, the computations appearing above have been relocated to auxiliary methods (`get_compounded_balance`, `get_lp_amount`,...) which perform them correctly without causing unnecessary precision loss.

M-3 Precision loss issues: double decimal conversion		
Severity: **Medium**	Impact: **Medium**	Likelihood: **Medium**
Category: Arithmetic		Files: price_provider.rs

Description

The double conversion in L#41-#45 and L#56-#60 of price_provider.rs passes via FixedI128 type (with fixed denominator 1e9) loses precision when the base asset decimals are >9:

```
        median_twap_price
            .mul_int(amount)
            .and_then(|a| FixedI128::from_rational(a,
10i128.pow(config.asset_decimals)))
            .and_then(|a| a.to_precision(self.base_asset.decimals))
            .ok_or(Error::InvalidAssetPrice)
```

```
        median_twap_price
            .recip_mul_int(amount)
            .and_then(|a| FixedI128::from_rational(a,
10i128.pow(self.base_asset.decimals)))
            .and_then(|a| a.to_precision(config.asset_decimals))
            .ok_or(Error::InvalidAssetPrice)
```

Recommendation

Convert the median_twap_price directly to the required precision.

Customer Response

Acknowledged and will fix.

Fix Review

The issue is resolved in commit 993fea5.

M-4 Centralization Risk

Severity: **Medium**	Impact: **Medium**	Likelihood: **Medium**
Category: Governance, Web2 security		Files:

Description

There is currently a single admin rule in the entire protocol which is allowed to do *everything* (e.g., upgrading the entire contract logic) but also required for fairly standard maintenance tasks (e.g., adjusting config parameters).

This is a dangerous situation from the point of view of web2 security which does not conform with standard security and risk management principles like separation of duties and least privilege access.

Recommendation

We recommend adding several less-privileged operational rules who will handle daily tasks like updating config parameters etc and reserving the admin rule to contract upgrades. We further suggest that all privileged rules would require multiple signatures.

Customer response

After adding RBAC to the protocol the compiled Wasm exceeded ~84KB (~74KB after Wasm optimization) which is beyond the current Soroban threshold of the "Ledger entry size (including Wasm entries) per Tx" (64 KB). So we had to revert it. We are planning to add RBAC after Soroban increases its limits. But as of now, we will either use the existing Multisig solution for admin or upgrade it in the future.

certora

M-5 There is no backup price feed		
Severity: **Medium**	Impact: **Medium**	Likelihood: **Medium**
Category: SEP-40 Oracle, Logic		Files:

Description

Like many other DeFi protocols, Slender requires precise off-chain price data for many of its critical functions (e.g., lending/borrowing, liquidations etc) which is accessed via an SEP-40 compatible oracle interface. However, it is by no means guaranteed that a given oracle feed would remain functional and correct forever. Unfortunately, while still a rare event, oracle failures are not unheard of. At best, unhandled oracle reverts can lead to a potential DoS. At worst, a malfunctioning oracle which reports a bad price could spell a disaster for the protocol. Thus, it is highly recommended for Slender to avoid putting all of its eggs in one basket by depending upon the correctness of a single external entity for its function.

Recommendation

We suggest including a fallback oracle in the protocol.

Customer response

Acknowledged and will fix.

Fix Review

Resolved in commit 993fea5.

M-6 There is no stale price check

Severity: **Medium**	Impact: **Medium**	Likelihood: **Medium**
Category: SEP-40 Oracle, Logic		Files:

Description

Slender does not check whether or not the information obtained via the SEP-40 oracle interface is stale (i.e., it does not check if the `timestamp` field of the `PriceData` struct is sufficiently recent).

Exploit Scenario

There are numerous possible situations in which this mistake leads to unwanted behavior which is bad for the protocol. For example, an attacker sets an automated script waiting for the moment when Slender's chosen SEP-40 oracle price update lags behind (e.g., due to outage). When such an event occurs, the attacker exploits this by taking an under collateralized loan at an incorrect price level, creating bad debt to the protocol.

Recommendation

Add a staleness parameter to the price feed config and logic to handle stale prices (when the price returned by the primary oracle is stale, use the backup oracle).

Customer response

Acknowledged and will fix.

Fix Review

The commit 993fea5 resolves the problem by adding a configuration parameter `min_timestamp_delta` and a test for staleness:

```
        if timestamp_delta > config.min_timestamp_delta {
            return Err(Error::NoPriceForAsset);
        }
```

Remark - note that if the timestamp is stale the chosen solution is to return an error, so the protocol would be unusable during such a time.

M-7 The protocol lacks circuit breakers (such as min/max prices)		
Severity: **Medium**	Impact: **Medium**	Likelihood: **Medium**
Category: SEP-40 Oracle, Logic		Files: price_provider.rs

Description

Many off-chain oracles (e.g., Chainlink) have internally configured min/max prices to prevent spurious reading. This however can be problematic in rare extreme events (e.g., flash crash, bridge compromise, or a stable coin depegging event). It is important for any lending/borrowing protocol to be able to recognize such outliers and install "circuit breaker" logic which checks if minAnswer <= reportedPrice <= maxAnswer.

Recommendation

Compute "sanity prices" off-chain and stop protocol action in cases of extreme price events.

Customer response

Acknowledged. Will fix

Fix Review

The issue is resolved in commit 993fea5 by adding min/max prices computed off-chain:

```
                    let is_sanity_price = twap_price >=
config.min_sanity_price_in_base
                        && twap_price <= config.max_sanity_price_in_base;
```

L-1 Some configuration parameters lack input validation

Severity: **Low**	Impact: **Medium**	Likelihood: **Low**
Category: Logic		Files:

Description

The function `set_ir_params` is a setter for the struct

```
pub struct IRParams {
   pub alpha: u32,
   pub initial_rate: u32,
   pub max_rate: u32,
   pub scaling_coeff: u32,
}
```

The following checks are performed by `require_valid_ir_params` -

1. initial_rate<= PERCENTAGE_FACTOR
2. max_rate > PERCENTAGE_FACTOR
3. scaling_coeff < PERCENTAGE_FACTOR

where PERCENTAGE_FACTOR = 10_000 (represents 100%). However, the following further checks should be performed:

4. initial_rate <= max_rate
5. scaling_coeff > 0

The function `set_initial_health` is a setter for the initial_health parameter which is treated as a percentage in the code. However, there are no sanity checks to verify it is indeed between zero and PERCENTAGE_FACTOR.

The function `set_flash_loan_fee` is a setter for the flash_loan_fee parameter which is treated as a percentage in the code. However, there are no sanity checks to verify it is indeed between zero and PERCENTAGE_FACTOR.

In addition, all these checks need to be performed in `initialize` (which is called upon pool deployment).

The function `set_price_feeds` gets as input the struct

```
pub struct PriceFeedConfigInput {
   pub asset: Address,
   pub asset_decimals: u32,
   pub feeds: Vec<PriceFeed>,
}
```

Where the elements of the feeds vector are

```
pub struct PriceFeed {
   pub feed: Address,
   pub feed_asset: OracleAsset,
   pub feed_decimals: u32,
   pub twap_records: u32,
   pub timestamp_precision: TimestampPrecision,
}
```

and

```
pub enum OracleAsset {
   Stellar(Address),
   Other(Symbol),
}

impl From<OracleAsset> for Asset {
   fn from(asset: OracleAsset) -> Self {
       match asset {
           OracleAsset::Stellar(address) => Asset::Stellar(address),
           OracleAsset::Other(symbol) => Asset::Other(symbol),
       }
   }
}
```

There are many obvious sanity tests that can be added here (e.g., between feed_asset and asset etc).

Recommendation

Add the extra sanity checks to validate the input of the setter functions and the initialize function called at deployment. In addition, it might seem reasonable to have some safety (as opposed to just sanity) boundaries in order to prevent some kind of a “fat finger error”.
The same is true for `configure_as_collateral` which does perform sanity checks but could potentially benefit from some safety checks.

Customer Response

Acknowledged. Will fix.

Fix Review

commit 993fea5 adds the necessary checks.

certora

I-1 Some of the names for the variables are misleading		
Severity: **Informational**	Impact:	Likelihood:
Category: Best Practice		Files: [account_positions.rs]()

Description

In L#[223-227]() of account_positions.rs the computation of *compound_debt* is reusing the name compound_balance:

```
        let compounded_balance = debt_coeff
            .mul_int(who_debt)
            .ok_or(Error::CalcAccountDataMathError)?;
        let debt_balance_in_base = price_provider.convert_to_base(&asset,
compounded_balance)?;
```

Recommendation

We suggest fixing it to improve the readability of the code.

Customer Response

Acknowledged and fixed.

I-2 Replace 10i128.pow with 10i128.checked_pow		
Severity: **Informational**	Impact:	Likelihood:
Category: Best Practice		Files:

Description

The power operation used in the code is not protected against overflow.

Recommendation

We suggest replacing it with its checked variant.

Customer Response

Acknowledged and fixed.

I-3 Some inline comments are inaccurate/outdated

Severity: **Informational**	Impact:	Likelihood:
Category: Best Practice		Files: rate.rs

Description

The inline comments for the function `calc_interest_rate` state:

```
/// Calculate interest rate IR = MIN [ max_rate, base_rate / (1 - U)^alpha]
/// where
/// U - utilization, U = total_debt / total_collateral
/// ir_params.alpha - parameter, by default 1.43 expressed as 143 with denominator
100
/// ir_params.max_rate - maximal value of interest rate, by default 500% expressed as
50000 with denominator 10000
/// ir_params.initial_rate - base interest rate, by default 2%, expressed as 200 with
denominator 10000
///
/// For (1-U)^alpha calculation use binomial approximation with four terms
/// (1-U)^a = 1 - alpha * U + alpha/2 * (alpha - 1) * U^2 - alpha/6 * (alpha-1) *
(alpha-2) * U^3 + alpha/24 * (alpha-1) *(alpha-2) * (alpha-3) * U^4
```

However this is not correct, as we an see from the code

```
    let num_of_iterations = if u > FixedI128::from_rational(1, 2)? {
        19
    } else {
        3
    };
```

We either use a binomial approximation with five terms (when alpha<=1/2) or twenty-one (when alpha>1/2).

Customer Response

Partially acknowledged - note that in the case where alpha<=1/2 , we actually do use four terms because we start from two in the cycle below. This was done for convenient tracking of terms. The problem is more the naming convention since `num_of_iterations` is actually not the real number of iterations… will fix.

I-4 Flash loan event sometimes include irrelevant info		
Severity: **Informational**	Impact:	Likelihood:
Category: Best Practice		Files: flash_loan.rs

Description

The event omitted in L#116 of flash_loan.rs

```
        event::flash_loan(
            env,
            who,
            receiver,
            &received_asset.asset,
            received_asset.amount,
            received_asset.premium,
        );
```

Includes the `received_asset.premium` which is not charged in case the user choose to borrow the assets following the flash loan (i.e., in this case the boolean invocation parameter borrow was set to true) which is misleading.

Recommendation

Add the value of the borrow parameter to the emitted event. In case `borrow = true`, set `received_asset.premium = 0`.

Customer Response

Acknowledged and fixed.

About Certora

Certora is a Web3 security company that provides industry-leading formal verification tools and smart contract audits. Certora's flagship security product, Certora Prover, is a unique SaaS product that automatically locates even the most rare & hard-to-find bugs on your smart contracts or mathematically proves their absence. The Certora Prover plugs into your standard deployment pipeline. It is helpful for smart contract developers and security researchers during auditing and bug bounties.

Certora also provides services such as auditing, formal verification projects,
and incident response.