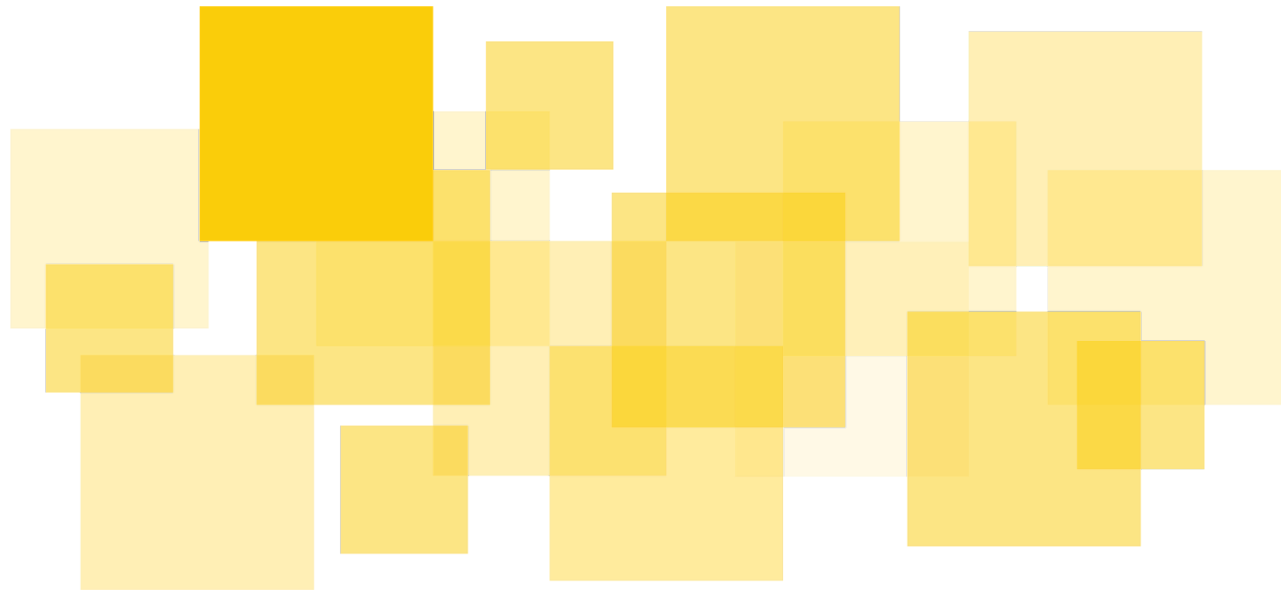


Security Audit Report

FxDAO Stellar

Delivered: May 17, 2024



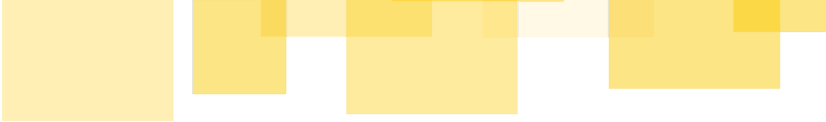
Prepared for FxDAO by





Table of Contents

- [Disclaimer](#)
- [Executive Summary](#)
- [Goal](#)
- [Scope](#)
- [Methodology](#)
- [Platform Logic and Features Description](#)
 - [Vaults](#)
 - [Stable Liquidity Pool](#)
- [Invariants](#)
- [Findings](#)
 - [\[A1\] It is Possible to Withdraw Rewards of Other Users and Prevent Them From Withdrawing Their Liquidity](#)
 - [\[A2\] Denial of Service Originated by Operations Using the Stable Liquidity Pool Share Price](#)
 - [\[A3\] Pre-calculating Parameters for Functions Calls Creates Liveness Issue](#)
 - [\[A4\] Protocol Allows Multiple Accepted Positions of Duplicate Index Vaults](#)
 - [\[A5\] Users May Pay More Than the Expected Protocol Fee When Managing their Vaults](#)
 - [\[A6\] Duplicate Currencies Based on Contract Address Can be Added to Vaults Contract](#)
 - [\[A7\] `liquidate` and `redeem` functions can be blocked by enormous unhealthy debt position](#)
 - [\[A8\] Vaults Can Be Misplaced In the Linked List of a Wrong Denomination](#)
 - [\[A9\] Vaults can be configured so that `opening_col_rate` < `min_vault_col`](#)
 - [\[A10\] No sanitation of `fee` for `vaults` or `stable-liquidity-pool`](#)
- [Informative Findings](#)
 - [\[B1\] Best Practices and Notable Particularities](#)
 - [\[B2\] User Can Deposit and Lock Deposits of Zero Tokens](#)
 - [\[B3\] It is Possible to Swap Assets In Liquidity Pools Without Paying Fees](#)
 - [\[B4\] Dependency on Governance Token Trust](#)

- 
- [B5] Contracts May Trigger Misleading Panic Messages
 - [B6] The Vaults Protocol Does Not Allow the Removal of Currencies
 - [B7] Only One Oracle is Used As Reference for Exchange Rates and No Validations Over The Fetched Values Are Performed
 - [B8] Stable Liquidity Pools have a Constant Exchange Rate
 - [B9] `protocol_manager` and `Admin` can be the same address
 - [B10] Rounding on Division Allow Users to Withdraw Dust From Liquidity Pools



Disclaimer

This report does not constitute legal or investment advice. You understand and agree that this report relates to new and emerging technologies and that there are significant risks inherent in using such technologies that cannot be completely protected against. While this report has been prepared based on data and information that has been provided by you or is otherwise publicly available, there are likely additional unknown risks which otherwise exist. This report is also not comprehensive in scope, excluding a number of components critical to the correct operation of this system. This report is for informational purposes only and is provided on an "as-is" basis and you acknowledge and agree that you are making use of this report and the information contained herein at your own risk. The preparers of this report make no representations or warranties of any kind, either express or implied, regarding the information in or the use of this report and shall not be liable to you or any third parties for any acts or omissions undertaken by you or any third parties based on the information contained herein.

Smart contracts are still a nascent software arena, and their deployment and public offering carries substantial risk.

Finally, the possibility of human error in the manual review process is very real, and we recommend seeking multiple independent opinions on any claims which impact a large quantity of funds.



Executive Summary

FxDAO engaged [Runtime Verification Inc.](#) to conduct a security audit of their contracts' code. The objective was to review the platform's business logic and implementation in Rust and identify any issues that could cause the system to malfunction or be exploited.

The audit was conducted over the course of 4 calendar weeks (March 28, 2024, through April 28, 2024) and focused on analyzing the security of the source code of FxDAO's Vaults and Stable Liquidity Pools contracts, which enables users to provide collateral backing off-chain assets in exchange for stablecoins, and also deposit and withdraw stablecoins to and from liquidity pools. These pools can be used to exchange assets that are held by it, providing the liquidity providers with a percentage of the traded amounts, charged as fees.

The audit led to identifying issues of potential severity for the protocol's health, which have been identified as follows:

- Functional correctness: [A1](#), [A2](#), [A3](#), [A7](#), [A8](#)
- Input validation: [A4](#), [A6](#), [A10](#), [B2](#)
- Potential threats to users' fund integrity: [A1](#), [A5](#), [A7](#), [A8](#), [A9](#), [B10](#)

In addition, several informative findings and general recommendations also have been made, including:

- Best practices and improvement suggestions: [B1](#), [B5](#), [B6](#)
- Code optimization-related particularities: [B1](#)
- Considerations on protocol trust: [B4](#), [B7](#), [B9](#)
- Notes on potential economic risks: [B3](#), [B8](#)

The client has addressed a substantial amount of the potentially critical findings and informative findings, and general recommendations have been incorporated into the platform's code. All of the remaining findings have been acknowledged by the client and deemed non-threatening to the integrity of the platform, when not intended by design.



Goal

The goal of the audit is threefold:

1. Review the high-level business logic (protocol design) of FxDAO's system based on the provided documentation;
2. Review the low-level implementation of the system for the individual Soroban smart contracts (Vaults and Stable Liquidity Pool);
3. Analyze the integration between the modules in the scope of the engagement and reason about possible exploitive corner cases.

The audit focuses on identifying issues in the system's logic and implementation that could potentially render the system vulnerable to attacks or cause it to malfunction. Furthermore, the audit highlights informative findings that could be used to improve the safety and efficiency of the implementation.



Scope

The scope of the audit is limited to the code contained in a private repository provided by the client. Within the repository and among other files, two contracts are highlighted as in the scope of the protocol. The repository and contracts are described below:

1. FxDAO-SC (Commit `47bc0803173702d4b9d88f58a94cbcbab9ebeaf0` , branch `main`): this repository contains several contracts designed for the FxDAO protocol. From it, two contracts were audited:
 - Vaults (`FxDAO-SC/vaults`): implements a lending protocol where users provide collateral to back assets being tokenized. The proposal of FxDAO is to initially collateralize off-chain currencies, issuing stablecoins reflecting their exchange rate to their collateral token;
 - Stable Liquidity Pool (`FxDAO-SC/stable-liquidity-pool`): implements the logic of an automated market maker for stablecoins. The exchange rate of the coins is fixed as 1-to-1, and the features of the contract allow users to deposit, withdraw, and swap tokens, with additional features related to the distribution of governance rewards. Users can obtain these rewards by locking their deposits for a period of time in which the governance rewards are distributed.

The comments provided in the code, and a general description of the project, including samples of tests used for interacting with the platform, and online documentation provided by the client were used as reference material.

The audit is limited in scope to the artifacts listed above. Off-chain, auto-generated, or client-side portions of the codebase, as well as deployment and upgrade scripts, are not in the scope of this engagement.

Commits addressing the findings presented in this report have also been analyzed to ensure the resolution of potential issues in the protocol.



Methodology

Although the manual code review cannot guarantee to find all possible security vulnerabilities as mentioned in our [Disclaimer](#), we have followed the approaches described below to make our audit as thorough as possible.

First, we rigorously reasoned about the business logic of the code, validating security-critical properties to ensure the absence of loopholes in the business logic. To this end, we carefully analyzed all the proposed features of the platform and the actors involved in the lifetime of deployed instances of the audited contracts.

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

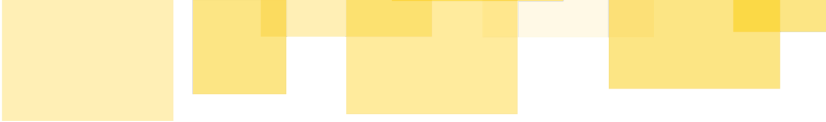
- Modeled sequences of mathematical operations as equations and, considering the limitations enforced by the identified invariants, checking if all desired properties hold for any possible input value;
- Manually built high-level function call maps, aiding the comprehension of the code and organization of the protocol's verification process;
- Modified and created tests to search and identify possible issues in FxDAO's smart contracts logic.

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

Furthermore, once the creation of higher-level abstractions and the process of understanding contracts was complete, attempts to use the protocol's business logic to break the collected invariants were performed, resulting in some of the findings presented in this report.

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

Any tests generated that could be of interest to the client have been provided to the client upon the delivery of this report.



Additionally, given the nascent Stellar-Soroban development and auditing community, we reviewed [this list](#) of known Ethereum security vulnerabilities and attack vectors and checked whether they apply to the smart contracts and scripts; if they apply, we checked whether the code is vulnerable to them.



Platform Logic and Features Description

The FxDAO's Vaults and Stable Liquidity Pool contracts compose a protocol built using the Soroban platform in the Stellar network. At the moment of writing this report, the protocol has already deployed contracts in testnet and has a documentation page available for users. These and other information about the protocol can be accessed through [FxDAO's landing page](#).

The following is the general description of each contract in scope of this audit engagement.

Vaults

The Vaults contract allows users to collateralize assets in exchange for tokens minted in the platform. In its current design, FxDAO's Vaults intend to collateralize off-chain currencies such as the American dollar (USD) and Euro (EUR). By collateralizing these assets, the users are rewarded with stablecoins where their value reflects the value of the asset in question. Each Vaults contract is responsible for handling all vaults that back assets using a single collateral token. This means that if a user wants to collateralize the USD currency using, for instance, XLM and BTC, two distinct Vaults contracts have to be created to do so. If a user wants to collateralize USD and EUR using XLM, a single Vaults contract can be used for the task, as different collateralized currencies (also referred to as denominations) can be handled in a single Vaults contract.

The way users can collateralize assets using the FxDAO protocol has the same high-level logic of a lending protocol: users interact with the contract providing collateral that will be held by the platform while receiving tokens in the form of issued debt. The collateral and data documenting the loan will remain on the platform until the debt is paid by its user or, in case the loan's health deteriorates (i.e., the value of the collateral falls in contrast to the value of the collateralized asset), its debt can be paid by other users through means of the `redeem` or `liquidate` operations. Vaults that back the same asset are placed in an ordered linked list, where the criterion of the ordering is the loan's economic health. Only the vaults that are placed in the first indexes of a linked list for a denomination or, in other words, the riskiest loans, can be liquidated, and only the riskiest vault for a denomination can be a target of the `redeem` operation. If in a position to be liquidated, the operation will be performed following the order from the lowest index to the highest.

Whenever creating a vault through the `new_vault` function or modifying its collateral/debt, its index in the denomination's linked list will be calculated using the following equation:

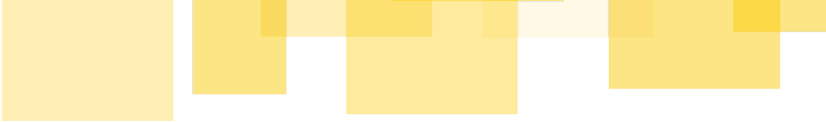
$$Vault.index = \lfloor \frac{1,000,000,000 \times vault_collateral}{vault_debt} \rfloor$$

Another way to look at the vault index is by observing it as a rate of collateral to debt, hence the first elements in a denomination's linked list have the lowest ratio, making them the most susceptible to characterizing an undercollateralized position.

If inserted in an empty denomination's linked list, this newly created vault will now be the one with the lowest index, and will not point to any other vault as its next list entry. If inserted in a populated denomination's linked list, unless it is inserted as the vault with the lowest index, the user has to provide a reference to the vault that will become its previous entry in the linked list. The next entry to this previous vault will now be the currently inserted vault's next entry, while the currently inserted vault will become the previous vault's next entry. Thus the denomination's linked list will be handled when modifying the collateral or debt of a vault.

The following pseudo algorithm describes the process of insertion of a vault in a linked list:

1. Let $Vaults_t$ represent the abstract sequence of vaults that are connected in the linked list at time t
2. Let $Vault$ be the representation of a vault being inserted in $Vaults_t$
3. Let $Prev_t$ represent the vault that will precede the inserted vault in $Vaults_t$ at time t
4. Let len_t be the length of $Vaults_t$.
5. **if** $Vaults_t - 1 = []$
6. **then** $Vaults_t = [Vault]$
7. **else if** $Vault.index < Vaults_t - 1[0].index$
8. **then** $Vaults_t = [Vault] + Vaults_t - 1[..]$
9. **else if** $\exists Prev_t - 1.next \implies True$
10. **then** $Vaults_t = Vaults_t - 1[0..i] + [Vault] + Vaults_t - 1[j..len_t - 1]; i + 1 == j; Vaults_t - 1[i].index \leq Vault.index \leq Vaults_t - 1[j].index$
11. **else** $Vaults_t = Vaults_t - 1 + [Vault]$



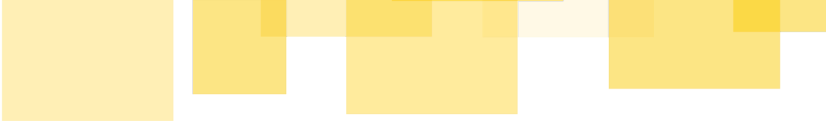
At the end of this operation, not only the vault should have been placed in its correct order according to its liquidation/redemption risk, but a reference to the vault with the lowest index, or higher liquidation risk, for that specific denomination should be updated. The value held by the `updated_lowest_key`, which should now point to the vault of the lowest collateral-to-debt ratio, can be described by the following expression:

$$updated_lowest_key = \text{if } Vaults_t = None \text{ then } None \text{ else } Vaults_t[0]$$

Whenever modifying a vault's collateral or debt, it will be removed from its denomination's linked list and will be reinserted, repeating the above-elaborated process.

Once created, a vault can have its collateral and debt manipulated by the following functions:

- `increase_collateral` : this function allows a user to send collateral to his vault, raising his collateral-to-debt ratio and, consequently, raising his vault index and potentially moving up the vault in the ordered linked list of vaults for its specific denomination;
- `increase_debt` : this function allows a user to issue more debt for a specific vault, the debt will be issued in the form of a token, which will be sent to the user. The collateral-to-debt ratio of that vault will drop, reflecting a reduction in the vault index and bringing the vault closer to a position where it can be redeemed/liquidated;
- `pay_debt` : this function allows a user to reduce his debt by burning the token initially issued as debt for his loan. The effect of this function on the user's vault is similar to the one seen in `increase_collateral`, raising the collateral-to-debt ratio. If the vault debt is paid partially, the debt will be reduced and the collateral will remain the same, but if paid in full, all the vault collateral will be sent back to the user and the vault will cease to exist;
- `redeem` : this function allows a user to pay for the debt of the vault of the lowest index for a specific denomination. In case the vault is over-collateralized, the calling user will receive a part of the vault collateral reflecting the value of the debt that is being paid, while the remaining collateral will be sent to the vault owner. Once the operation is done, the vault will cease to exist;
- `liquidate` : this function allows a user to pay for the debt of the vault of the lowest index for a specific denomination. The vault targeted by this operation must have the rate of its collateral value to its debt value beneath a minimum defined rate, and this rate is defined per denomination. The caller will receive the total vault collateral minus a fee, and once the operation is done, the vault will cease to exist. This function can be called to liquidate



multiple vaults in the same denomination, but all must be at the beginning of the ordered linked list of vaults for that particular denomination.

All operations that modify the amount of collateral in a vault will pay a fee, which is deducted from the collateral amount modified and sent directly to the protocol treasury address. The fee is always calculated using the following formula:

$$fee = \left\lceil \frac{amount \times core_state.fee}{10,000,000} \right\rceil$$

Where *core_state* represents the storage element of the contract which is responsible for keeping the core protocol information, and *fee* represents the stored fee value for the Vaults contract (10^6 representing 100%).

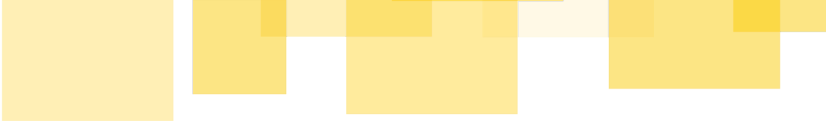
The fees obtained through liquidations are not sent to the treasury, instead being kept in the contract as an approach for mitigating the possibility of protocol insolvency in the future.

An instance of a Vaults contract has an administrator and a protocol manager. They are capable of creating denominations in a vault, which are needed in order to create loans backing the asset represented by that denomination. Besides that, each administrative figure is capable of using a set of operations controlling the protocol, which includes setting parameters for the vaults' operations (e.g., fee value), activating a panic mode feature, which prevents operations from happening in case of an emergency, and upgrading the Vaults contract altogether.

Stable Liquidity Pool

The FxDAO protocol also aims to provide a platform where users can swap stablecoins that reflect the same currency. This will be done through instances of the Stable Liquidity Pool contracts. These contracts work as automated market makers with a fixed exchange rate of 1-to-1, where users are able to supply assets to these contracts, also referred to as "pools". The assets that have been supplied work as liquidity for trading operations, where a user can call functions of pools requesting to exchange a number of tokens for the same number of tokens of a different stablecoin (that reflects the same currency).

During the creation of a stable liquidity pool, its creator must specify, among other things, a list of assets that the pool will be able to handle. This list has no upper or lower bounds for length enforced onto it. Also, other information is provided at the instantiation of an instance of this



contract, such as the governance token address that will be distributed as user rewards, the fee charged on swap operations, the treasury address, and other managerial values used by the protocol.

The features that enable deposits, withdrawals, and swaps in stable liquidity pools are relatively straightforward, but with special remarks that differentiate them from other automated market makers for stablecoins. For a start, a user can perform swap operations from one asset to another held in the pool, and the exchange rate between these assets is always 1. This is based on the assumption that Stable Liquidity Pool contracts will only be configured to hold assets that represent the same currency. For instance, if a pool for USD assets is created, it would hold only USD stablecoins (such as USDC, USDT, and USDx).

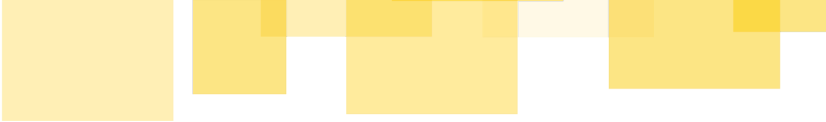
Also, no fees are charged when depositing or withdrawing assets from the pool. Due to this, all deposits performed by users must remain in the contract for at least 2 days. This locking mechanism was designed to prevent users from atomically depositing and withdrawing assets from the pool, consequently opening the possibility of what is essentially a swap of assets with no cost in fees.

One of the consequences of this mechanism is that, if a user already has a total of X tokens deposited in the pool that is currently unlocked (meaning that two days have already gone by since this initial deposit), if this user carries out a deposit of Y tokens, his total balance of $X + Y$ tokens will now remain locked in the contract for two days.

Furthermore, when users perform deposits, shares will be issued to them to represent their position in the liquidity pool. These shares, unlike the issued debt in the Vaults contract, are not produced in the form of tokens, instead being a number kept in the persistent memory of the smart contract. The amount of shares issued to that user on a deposit is defined by the following equation:

$$shares_to_issue = \left\lfloor \frac{amount_deposited \times 10,000,000}{core_state.share_price} \right\rfloor$$

Once again, *core_state* is the object that holds managerial values of the protocol, and *share_price* is the rate at which many shares are issued to a user according to how many tokens have been deposited in a single operation. The value of *share_price* is initially defined during the instantiation of the contract, and starts with the value of 10^6 , to represent a 1 with 7 decimal places. *share_price* can only be modified under two circumstances:

- 
1. During swap operations, where the fee kept as pool profit reflects an increase in the total amount of tokens held by the protocol with no deposit;
 2. During a withdrawal operation where all of the liquidity of the pool is removed, resetting the share price back to 10^6 .

The growth during swap operations happens as a consequence of how fees are managed on swaps, where the total fee amount is calculated by the following:

$$fee = \left\lceil \frac{amount \times core_state.fee_percentage}{10,000,000} \right\rceil$$

The resulting fee is now split into two values: half (rounded up) is transferred to the protocol treasury and half (rounded down) remains in the pool as pool profit. The share price is now updated by taking into consideration this progression in the total amount of liquidity in the pool resulting from a swap operation, and its updated value is calculated as follows:

$$new_share_price = \left\lfloor \frac{new_total_deposited \times core_state.share_price}{core_state.total_deposited} \right\rfloor$$

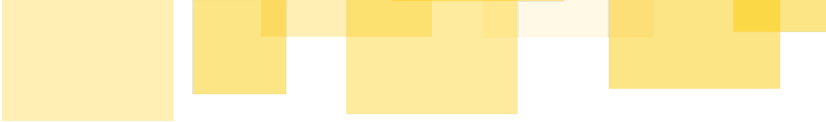
new_total_deposited being the new total balance of the pool after the swap (*core_state.total_deposited* + *pool_profit*), and *core_state.share_price* right after the swapping operation, receiving now the value of *new_share_price*.

The more swaps are performed (and thus the more pool profit there is), the lesser the number of issued shares on deposits.

Besides the `deposit`, `withdraw`, and `swap` functions, other functions are available for user-contract interaction. These functions are `lock`, `unlock`, and `distribute`, which are related to the distribution of governance tokens by the protocol.

Once users have deposited an amount of liquidity into a stable liquidity pool, they have the option of locking their deposit for a minimum period of 7 days. This commitment is done by calling the `lock` function, and this will prevent a user from withdrawing his assets until this minimum time window has been expended, and the `unlock` function is called by this same user.

When locking their assets, the user will also keep in the persistent memory of the contract a factor, also referred to as a rate of distribution, at the time of the `lock` function execution. This



"snapshot" of the current distribution rate will later be used to calculate his rewards when unlocking the deposit.

While the governance platform is outside this audit engagement's scope, the FxDAO team shared information about the intended methodology for issuing and distributing its governance tokens. In relationship to the Stable Liquidity Pool contract, governance tokens should be sent periodically to it through means of the `distribute` function.

The `distribute` function can be called from any address and is responsible for making a deposit from the caller to the contract. The deposit amount is a parametrized number of governance tokens. When depositing governance tokens to the Stable Liquidity Pool contract, the factor of distribution used to calculate the rewards of each individual user is updated. The calculation is performed as follows:

$$locking_state_t.factor = locking_state.factor_{t-1} + \left\lfloor \frac{amount \times 10,000,000}{locking_state_{t-1}.total} \right\rfloor$$

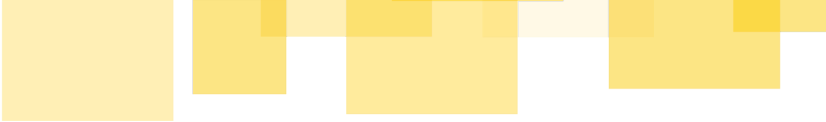
Where *amount* represents the number of governance tokens deposited to be distributed, and *locking_state* represents the stored object used to represent the information about locked deposits and necessary data for calculating how many governance tokens each user should receive. Within *locking_state*, a *factor* is used for the calculation of rewards for individual users, and also a *total*, representing the sum of liquidity of all locked deposits.

With deposits locked into the pool and rewards sent to it, users who unlock their assets from this pool by calling the `unlock` function will be rewarded with a fraction of the total governance tokens held by the contract. The more a deposit remains locked while rewards are being constantly distributed, the more rewards the user will be awarded with, as the amount of governance tokens sent to the user is calculated using the following equation:

$$reward = \left\lfloor \frac{deposit.shares \times (locking_state_t.factor - deposit.snapshot)}{10,000,000} \right\rfloor$$

deposit.shares indicating the number of shares issued to the user's liquidity for that locked deposit and *deposit.snapshot* representing the factor of rewards distribution at the moment the deposit was locked.

Once the deposit has been unlocked, the user can withdraw his liquidity from the pool.



The Stable Liquidity Pool contract can be upgraded and has no other administrative functions. The configuration of the pool properties such as fee percentage, governance token address, and treasury address remain the same from the moment it has been instantiated and have no means of being modified unless the contract is upgraded.



Invariants

During the audit, invariants have been defined and used to guide part of our search for possible issues with both Vaults and Stable Liquidity protocols. With the help of the FxDAO team, the following invariants were gathered:

Vaults

- **[VA] The contract takes priority when there are “decimals”:** anytime a number is rounded, the operation must be done aiming to benefit the protocol. For example, if the protocol is receiving funds, possible divisions over the value to be received are rounded up while the opposite happens if the protocol is sending funds;
- **[VB] The vaults are always ordered by the index from lowest to highest:** This means that the riskier the vault, the lowest its position in the list;
- **[VC] Redeem operations are always performed from the riskier vault and there shouldn't be any loss in value:** The one who is redeeming stablecoin should get the same value in collateral while the original owner of the vault should get the rest of the collateral, meaning that if the vault was at a 119% ratio, then the one calling the function gets 100% while the owner of the vault the other 19%. The vault needs to be removed after that;
- **[VD] Index of the vault should always be calculated after fees:** for example, if there is a fee involved in the transaction, the protocol first reduces the collateral by the fee amount, and then calculates the index with the new collateral amount;
- **[VE] When the protocol is in panic mode, it shouldn't be possible to issue new debt;**
- **[VF] No vault should be updated below the open ratio (the protocol uses two ratios: open debt ratio and liquidation ratio):** The protocol requires users to open their vaults with a 115% collateral ratio while the liquidation ratio is at 110%, the idea is that if a user is increasing its debt of existing vaults, it should not be possible to be below 115% when doing that;
- **[VG] Liquidations should always start at the lowest vault:** If there are two vaults in liquidation status (IE they are at a lower rate than 110%), the protocol must keep the order when liquidation them according to their position in the Vaults linked list;

- **[VH] When inserting, removing, or updating a vault, the linked list should always be sorted from riskier to safer;**
- **[VI] Both liquidations and redemption operations are considered full-value transactions:** this means that if a Vault has a debt of \$1000, then the liquidation/redemption will be for that specific amount IE the caller of the function needs to be able to pay the full amount of debt.

Stable Liquidity Pools

- **[SLPA] The contract takes priority when there are “decimals”:** Similar to what is done in the Vaults contract **[VA]**;
- **[SLPB] Swaps should always be at 1:1:** Assets in the pool are always swapped at the same price no matter what happens outside of the pool (it's the risk the depositor assumes);
- **[SLPC] When withdrawing a deposit, the withdrawal should include the profit from swaps (share price increased);**
- **[SLPD] A deposit needs to have a locking period no matter if the depositor wants to lock it for a longer time to receive the governance token.**

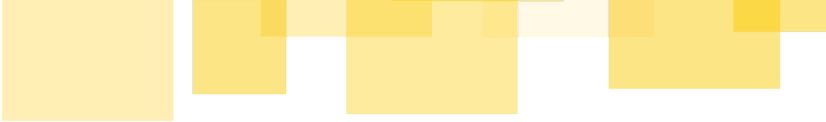
Additionally, a few other invariants have been generated internally. These invariants are:

Vaults

- **[V1]** There cannot be duplicated currencies registered based on `denomination` ;
- **[V2]** There cannot be duplicated currencies registered based on `contract` ;
- **[V3]** All functions that reduce debt during `panic mode` should be available;
- **[V4]** Inactive currencies cannot have debt created or removed, except by `liquidate` operations;
- **[V5]** The head of the linked list (`lowest_key`) always points to the vault with the lowest vault index;
- **[V6]** No vault can exist outside of the linked list;

Stable Liquidity Pools

- **[SLP1]** The reward between times $(t_n, t_m]$ for a constant `shares` is deterministic;
- **[SLP2]** The reward is earned only for shares inside the protocol for the time in which those shares are in the protocol and the rewards have not been claimed;

- 
- **[SLP3]** The total deposited value by the user can only be withdrawn after two days since the last deposit from that same user;
 - **[SLP4]** Users who lock their deposits for longer receive more rewards than those who lock their deposits for a shorter amount of time.

After analysis, we believe that some of these invariants hold and some of them do not. We have commented on the following invariants (if an invariant is not listed, we have reasoned that it holds and have no comment):

- **[VB]** This invariant holds, however, there is non-deterministic ordering for duplicate indices. See finding **[A4]**.
- **[SLPA]** FAILS. Incorrect rounding for *new_share_price* in **swap** function. See finding **[A10]**.
- **[V1]** This invariant holds, except there is no normalization of capitalization. See finding **[A6]**.
- **[V2]** FAILS. The same address can be used for multiple denominations. See finding **[A6]**.
- **[SLP4]** FAILS. Users can receive rewards for a longer time than is fair. See finding **[A1]**.

FxDAO has acknowledged the comments and corrected the protocols where appropriate so that the failing invariants now hold.



Findings

Findings presented in this section are issues that can cause the system to fail, malfunction, and/or be exploited, and should be properly addressed.

All findings have a severity level and an execution difficulty level, ranging from low to high, as well as categories in which it fits. For more information about the classifications of the findings, refer to our [Smart Contract Analysis](#) page (adaptations performed where applicable).



[A1] It is Possible to Withdraw Rewards of Other Users and Prevent Them From Withdrawing Their Liquidity

Severity: Medium

Difficulty: Low

Recommended Action: Fix Code

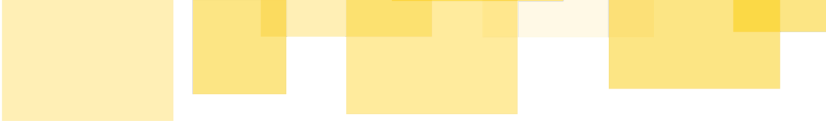
Addressed by client

Description

The stable liquidity pools of the FxDAO protocol allow users to lock their liquidity in these smart contracts in exchange for receiving protocol governance tokens. The core logic of the distribution of these tokens benefits users that maintain their liquidity locked for longer, and the minimum time in which the liquidity must remain locked to give rights of receiving governance tokens to a user is 7 days. In other words, whenever a user locks his liquidity in a stable liquidity pool, the `Deposit` object responsible for keeping track of the user deposit information will be updated with a snapshot of the rewards distribution factor and a timestamp recorded on the `unlocks_at` variable. `unlocks_at` defines when a deposit can be unlocked (if it is locked) or withdrawn (if it is not locked). The distribution factor, in turn, continues to grow while rewards are distributed by the protocol, and users with a lower recorded snapshot of this factor will receive more rewards, meaning that the earlier you lock your deposit, the more rewards you receive.

The stable liquidity pool contract has an issue that allows users to circumvent this waiting time of locked deposits, and it can be achieved simply by depositing once more after locking your deposit. Since no validations are preventing locked deposits from being modified, the `unlocks_at` variable will now be overwritten to reflect the protection guardband of a deposit for only 2 days, instead of 7. Furthermore, the user's issued shares will also be updated in this operation.

It is important to highlight that, in the current contract version, shares can be issued to users who have locked deposits. This is especially dangerous considering that **the number of shares of a locked deposit is used to calculate the amount of token rewards that will be received by that user when unlocking**, together with the rewards distribution factor, and the **distribution factor is calculated using the number of shares of the user at the moment he locked his deposit**. This implies that a user who locks his liquidity in the protocol and, afterward, deposits more assets into the pool, will have his rewards calculated without



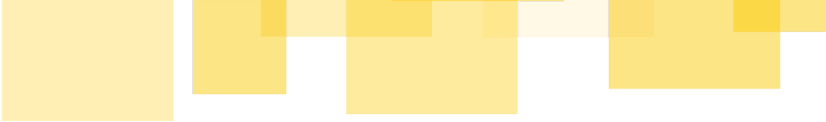
considering the shares issued after locking his deposit, and the user will be able to withdraw rewards that were meant for another user.

The last consequence of this combination of issues is that once a malicious user withdraws rewards that were kept for the rightful owners, if said owners attempt to unlock their deposits and withdraw their rewards, the contract logic will panic due to an attempt to transfer tokens that the contract does not hold. The honest users who locked their liquidity in the protocol now cannot withdraw their assets.

Scenario

To clarify this finding, the following scenario was elaborated to demonstrate how the presented issues can be exploited:

1. Alice is a malicious actor and joins the stable liquidity pool as soon as it is deployed. She joins the pool by depositing 0 amounts (as seen possible in finding [B2](#)) of an asset of that pool (or a negligible amount of the asset);
2. Right after joining, Alice locks her deposit. Because of the protocol logic, we register the distribution factor at the time, and not how much she proposed to lock. The amount of shares added to the total locked here has no particular meaning to the scenario;
3. This step happens during a period of weeks or months. Regular users join the pool and use it following its intended design. They provide liquidity, lock their deposits, and swap assets, while Alice maintains her assets somewhere else. Governance token distribution happens regularly, increasing the governance token distribution factor;
4. When Alice judges that the locking state factor grew sufficiently for her liking, she deposits all her assets into this pool aiming to raise her shares as much as she can. Her `unlocks_at` now points to 48 hours in the future;
5. After waiting 48 hours, Alice calls the unlock function and, even though she only committed the real value of her deposit to the protocol governance for two days, she will receive governance tokens as if she locked her deposit since the beginning of the existence of the protocol. Furthermore, the users who followed the protocol design, locked their full deposits, and waited for their rewards will now be deprived of their rightful governance tokens;
6. If Alice manages to grow her shares enough to withdraw all governance tokens from the pool;

- 
7. The subsequent calls of rightful owners to the unlock function will result in panic due to attempts to withdraw governance tokens that the pool does not hold anymore. Furthermore, these users will not be able to withdraw their provided liquidity, as their deposits will be considered locked.
-

Recommendation

Prevent users from circumventing the mandatory waiting time needed to participate in the governance token distribution by preventing locked deposits from being modified. Otherwise, keep track of the status of the deposit of the user when locked, and use the number of shares that the user had at the time to calculate how many governance tokens he should receive as a reward.

Status

This finding has been addressed by enforcing that locked deposits cannot be modified (commit id `d56f0dee08a4bee907cab69f80e3ec5936edbd37` on the branch `audit-fixes`).



[A2] Denial of Service Originated by Operations Using the Stable Liquidity Pool Share Price

Severity: Medium

Difficulty: High

Recommended Action: Fix Code

Addressed by client

Description

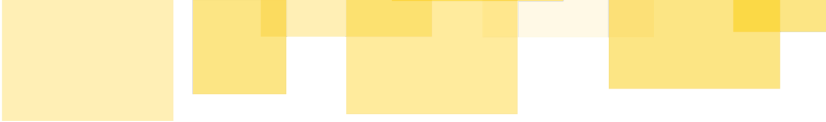
For liquidity pools in general, one of the riskiest moments regarding its economic safety comes right after its deployment. The first depositor of liquidity, depending on the algorithm for the distribution/issuance of shares and for the modification of share price during swaps, will have significant power over the pool depending on how much he deposits and what operations are performed in the pool whilst he is the only one participating in it.

In FxDAO's Stable Liquidity Pool scenario, given that you can't perform operations over your shares outside of managing them within the protocol, many of the risks are mitigated. Yet, one remains: the `share_price`, used to represent how much the share of a user is worth, moves only in one direction unless all liquidity is removed from the protocol, and this may carry safety implications.

Considering the contract's logic, there is a way of accelerating the rate at which the `share_price` grows, and it is by making swaps where the division of `new_total_deposited` (which represents the total amount of assets held by the pool after a swap) by `core_state.total_deposited` (representing the total amount of assets held by the pool before the swap) is the biggest possible. In case multiple consecutive swaps are performed, each iteration of the swaps will cause the share price to grow and, if large enough, this can cause a denial of service for the `deposit` and `swap` functions, as multiplications over the `share_price` variable may panic due to a `u128` overflow.

Scenario

To illustrate this, if we envision the stable liquidity pool contract using a fee of 0.2%, if the user deposits and swaps 10^3 , protocol profit (half of the total fee) will exist, generating a 0.1% increase of the `share_price` variable per swap. If swaps of the total value of the pool are



repeated roughly 72642 times, the swap itself will not be able to be performed anymore, as `share_price` will exceed the maximum value for a u128. Given that operations such as the multiplication of the deposited amount by the share price happen when calling the `deposit` function, the `share_price` variable doesn't have to be as close to `u128::MAX` for deposits to panic, meaning that the number of operation needed to cause a denial of service for the deposit function doesn't have as high as presented in this scenario.

While this scenario might seem a little bit far-fetched, its alternative is that this happens naturally with user trades within the lifetime of the smart contract if, for some reason, a tiny amount of assets is kept at the pool during its lifetime, but users keep exchanging high amounts of the pool's assets.

Recommendation

A way to mitigate this is to require that, when the pool has no liquidity in it, the next user to join the pool should be forced to provide a slightly higher value in his deposit. In the scenario that has been provided, if he deposits 10^4 of a token instead of 10^3 , which is still a small value if we refer to stablecoins, the rise of the `share_price` will be much slower, at 0.01% per swap operation instead of 0.1%. This greatly reduces the possibility of this issue manifesting within the lifetime of the FxDAO Stable Liquidity Pool protocol.

Status

This finding has been addressed by enforcing a minimum amount for deposits, while also requiring that, once a pool receives liquidity, a minimum amount of liquidity will remain in the pool at all times (commit id `75f88ea6588973a386deb26d64b31ca6288efcdc` in the branch `audit-fixes`).

[A3] Pre-calculating Parameters for Functions Calls Creates Liveness Issue

Severity: Low

Difficulty: Medium

Recommended Action: Fix Design

Not addressed by client

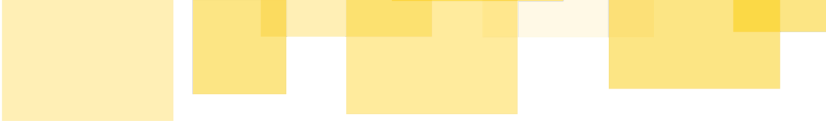
Description

Considering the logic of the Vaults contract, user loans taken in the protocol are managed through vaults. Each user that provides collateral and takes a loan now has an instance of a vault containing the operation information, and the set of instances of all users is managed through linked lists. These linked lists are ordered such that the entry with the lowest index represents the vault with the highest risk of liquidation, also allowing it to be a target of a `redeem` operation.

When new vaults are created, they are added to their linked list according to the denomination of the stablecoin and issued token contract address. The list maintains its ordering based on the calculated vault index. To successfully insert into this list, the user must precalculate the index and provide the key of a vault that precedes the position the new vault will take. Since there is no guarantee that the state on-chain will be the same as the state observed and used in the calculations off-chain, the scenario is possible where users correctly calculate their position but another transaction is interleaved before theirs that invalidates their previous calculation. A concurrent timing table to represent the scenario described above is:

Time	User 1	Vaults	User 2
t1	read vaults	A B	read vaults
t2	offchain calc vault C: A < C < B	A B	offchain calc vault D: A < D < B (NOTE: C < D)
t3	new_vault(index: C, prev: A)	A C B	
t4		A C B	FAILS: new_vault(index: D, prev: A)

User 2's attempt to add the vault will fail as the previous key should of been C, but C was not entered at the time they read the state.



If the system gets high traffic and there is a clustering of index values, there could be a scenario where a transaction is repeatedly rejected due to this scenario.

This problem also appears in the `stable-liquidity-pool` contract for the `withdraw` function. The user precalculates the value of `shares_to_redeem` off-chain, and that value is used in an on-chain calculation for `calculated_amount_to_withdraw`, and compared with `withdraw_amount` which calculates the same value but is not dependent on `shares_to_redeem`. The issue presents itself again, as if the user reads the state and calculates `shares_to_redeem` correctly, another transaction can be interleaved which will change the `total_shares` of the protocol, and the transaction will fail as `calculated_amount_to_withdraw` will differ from `withdraw_amount`.

Furthermore, this issue can manifest in any of the other functions in both Vaults and Stable Liquidity Pool contracts where parameters have to be calculated prior to a function call.

Recommendations

The ideal solution for such issues is to prevent users from submitting data that the contract can infer, but this may create cases where there is a divergence between what could be implemented and the intended contract design.

A partial mitigation would be to take the `prev_key` parameter as a hint and have logic to find the correct position by traversing the Linked List in the correct direction. Although there is the potential for the traversal to have $O(n)$ complexity where n is the number of vaults in the Linked List, this would assume that a completely incorrect hint was provided. In practice, if called correctly, the position of the correct index is likely to be close to the index provided. However, this assumes that `prev_key` still points to a valid vault, which may not be the case if the vault is removed, liquidated, or redeemed.

The most robust solution would be to calculate the correct position on the chain, in which case a Linked List is likely not the best data structure for the design due to the linear time complexity for most operations in the worst case. Instead, a Binary Tree would provide $O(\log n)$ complexity for inserting and removing a vault, without needing prior calculation, and no possibility of the transaction being rejected due to providing a stale `prev_key`.

Description



This finding has been acknowledged by the client.

[A4] Protocol Allows Multiple Accepted Positions of Duplicate Index Vaults

Severity: Low

Difficulty: Low

Recommended Action: Fix Design

Not addressed by client

Description

As described in the [Platform Logic and Features Description](#) section and in finding [A3](#), vaults are managed in an ordered linked list where the first entry represents the vault with the highest risk of liquidation and redemption.

With that in mind, new vaults are added to a linked list where the head is stored in `VaultsInfo` object. This object is responsible for keeping the information of the vaults linked list for a specific denomination-stablecoin contract address pair. The starting point of this linked list, which is the vault with the lowest index is stored in the field `lowest_key`. An invariant for the protocol is that all vaults in this linked list are ordered from lowest to highest vault index ($\lceil \frac{10^9 \times total_collateral}{total_debt} \rceil$). Calculating the indices using this ratio means that collisions between vault indices are possible (2 or more `Vaults` have the same vault index). Given 2 `Vault`s A and B , where $A \neq B \wedge A.index = B.index$, both $\dots \rightarrow A \rightarrow B \rightarrow \dots$ and $\dots \rightarrow B \rightarrow A \rightarrow \dots$ are permissible orderings. The protocol does not have any logic to handle collisions and will accept both cases which can be selected by a user providing the appropriate index to add their `Vault` to.

Having the ordering towards the back means that there is less (perhaps only marginally) of a chance of being liquidated or redeemed, as the first vaults in the linked list are liquidated and redeemed first.

Tests validating this finding have been produced and delivered to the client with this report.

Recommendation

Whenever creating or modifying a vault, when inserting the vault in the ordered linked list, if there is a collision of indexes among vaults, place the last vault to be inserted as the first one among the vaults with the same index.

Status



This finding has been acknowledged by the client.

[A5] Users May Pay More Than the Expected Protocol Fee When Managing their Vaults

Severity: Low

Difficulty: Low

Recommended Action: Fix Code

Addressed by client

Description

Considering the Vault contract, all operations that perform modifications over the value of a vault's collateral have to pay fees to the protocol treasury. For the sake of the economic safety of the pool, division operations involving the calculation of fees are always rounded up, preventing loss due to precision.

Another characteristic of the functions that modify the collateral amount of vaults is that there is no lower bounds on the deposited amount for increasing collateral. Using the `new_vault` function as an example, which, as the name suggest, is the function called when creating a new vault, the fee paid by the user is an output of the following operation: $\lceil \frac{DA \times Fee}{10^6} \rceil$, where DA represents the deposited amount, and Fee represents the fee percentage (10^6 being 100%).

If the user deposits anything below $10^6 / Fee$, the result of the division for obtaining the fee will be rounded up to 1, resulting in a fee value that is more than the expected percentage of the fee. Let's assume that we have a vault where the fee is 0.1%, so it would be represented by 10^3 , and the user deposits a value lower than 10^3 . 1 will always be more than 0.1% of whichever value is below 10^3 and this may be problematic if the vault is using a high-valued token with low precision on its decimals as collateral for a denomination.

Recommendation

To prevent unfair loss to the users, enforce that deposits of any value lower than $10^6 / Fee$ will fail.

Status

This finding has been addressed by enforcing that the amount for deposits is greater than the fee (commit id `f52959f262200237953ddf28ee889597f83b32b9` in the branch `audit-fixes`).



[A6] Duplicate Currencies Based on Contract Address Can be Added to Vaults Contract

Severity: Low

Difficulty: Medium

Recommended Action: Fix Code

Not addressed by client

Description

In the FxDAO vault protocol, a `Currency` is an object used to represent an asset that is being tokenized. It is composed of the asset symbol (coded as `denomination:Symbol`), the address of the token to be minted when collateralizing this asset (coded as `contract:Address`), and a boolean that indicates if the operations over that currency can or not be performed in that contract.

To illustrate how this works, let's say that we would like to use FxDAO's vault contract to tokenize the United States of America dollar. The first step to do so would be to call the `create_currency` function of the contract providing the symbol of that currency, which would be `USD`, and the contract address that will be minted when providing collateral for this currency.

Given the explanation above, we highlight there is logic to stop duplicate assets from being added based on `denomination`, however, no logic exists to prevent a duplicate from being added based on the minted token contract. For example, `usd` and `USD` could both be added as currencies using the same minted token contract address, as well as using symbols that could represent completely different currencies.

Furthermore, this also allows two denominations, one for the US dollar `USD` and another for Euro `EUR`, to be created with the same minted token address.

Tests validating this finding have been produced and delivered to the client with this report.

Recommendation

The protocol should account for the scenario where new currencies could be created using different symbols, but the same minted token address.



Status

This finding has been acknowledged by the client.

[A7] `liquidate` and `redeem` functions can be blocked by enormous unhealthy debt position

Severity: Medium

Difficulty: Medium

Recommended Action: Fix Design

Not addressed by client

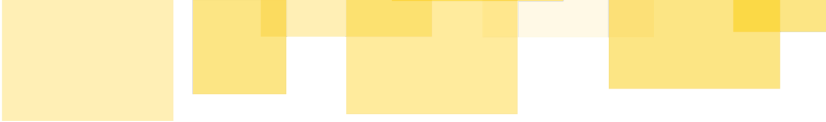
Description

The `vaults` contract relies on debt positions being entered in a linked list (*vaults*) which is ordered based on their health ratio of collateral to debt ($index = \lfloor \frac{1,000,000,000 \times vault.collateral}{vault.debt} \rfloor$). Two methods of the `vaults` contract `liquidate` and `redeem` can only be called from the head of the linked list, so it is impossible to skip the head and call either of these functions on vaults after the head. This means that in the event that the lowest health vault cannot be liquidated or redeemed, these functions will be blocked on that vault and unable to be called at all. What aggravates this detail is the fact that redemption and liquidation operations can only be performed by paying the full debt of a vault, leading to what we elaborate below.

Consider a debt position *vault1* which has the highest debt of all the vaults, and also the lowest index. Formally, $\forall vault \in vaults. vault \neq vault1 \implies vault1.index < vault.index \wedge vault.debt < vault1.debt$. Given that *vault1* is a maximum on debt, it is true that no other user can liquidate or redeem *vault1* using their vault's debt alone. Therefore, in order for another user with a *vault* to liquidate or redeem *vault1*, they must acquire *K* debt tokens, where $K = vault1.debt - vault.debt$. For a user without a vault, $K = vault1.debt$.

There are methods for users to acquire *K* debt tokens, some being creating a new vault or increasing their collateral and debt sufficiently, swapping for the debt token through exchanges, and using flash loans. The problem is that the higher *K* becomes, the more friction is introduced in users being able to acquire *K* debt tokens. This may mean that for extremely large *K*, there is a large amount of time for users to acquire *K* debt tokens. During this time the functions `liquidate` and `redeem` are unusable.

Flash loans should allow smaller users to be able to liquidate larger positions as they do not need to have access to the collateral required to acquire *K* themselves. However it shifts the



bottleneck to accessibility to flash loans, the lenders having sufficient liquidity.

Recommendation

Initially, a strategy for mitigating this issue is introducing the capability to partially liquidate an unhealthy loan, which currently is not supported in the FxDAO vaults protocol.

Status

This finding has been acknowledged by the client.



[A8] Vaults Can Be Misplaced In the Linked List of a Wrong Denomination

Severity: Medium

Difficulty: Medium

Recommended Action: Fix Code

Addressed by client

Description

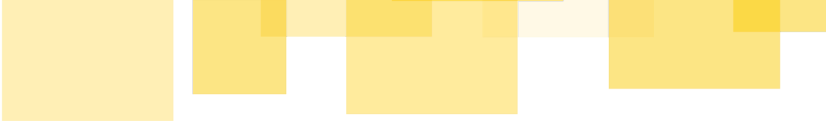
Whenever vaults are created or modifications are performed on them, the caller has to provide a reference to the vault which will be placed right behind them in the linked list ordering. The only criterion for this ordering is the previous vault's index. The core of this issue is that, throughout the validations performed over the previous vault and its relationship with the currently modified/created vault, there are no validations that enforce that `prev_key` (which identifies the previous vault) has the same denomination as the current vault. In other words, I can use a `prev_key` referencing a vault that is backing a currency different from the one that my vault is, so long as I respect the rules over the positioning of the index. Such misplacing may happen under the condition that the inserted vault is not placed as the highest-risk vault in that linked list (i.e., it isn't placed as the first element of the list).

It also opens room for unexpected behaviors for the redeem and liquidate operations if such a vault is at risk, as the wrong exchange rate will be used for the calculations of the collateral to be withdrawn potentially benefitting the user who had his vault redeemed.

Furthermore, if this misplaced vault is eventually liquidated or redeemed, the variables used for representing the amount of assets held by the protocol will no longer reflect the correct value and may cause problems to the protocol in the long run (e.g., an underflow causing a Denial of Service when trying to liquidate/redeem/pay debt).

Scenario

One case that exemplifies how this can be exploited is where you have two denominations for vaults; one backing an asset that is more stable than the other. For the sake of the example, let's say that one vault backs USD as collateral while the other backs EUR in a scenario where USD is more stable when compared with EUR. A malicious user can create a vault giving the denomination for EUR, but providing as the `prev_key` a vault that backs USD. If the user calculations of the indexes are indeed correct, the EUR vault will be inserted into the linked list



reserved for USD vaults, while its information will be added to the linked list which refers to the EUR denomination. Given the nature of price shifts for USD in comparison to EUR in this specific scenario, the user's vault will be in a safer position by being misplaced in the wrong linked list, providing him with an unfair advantage.

Recommendation

Whenever performing operations over the position of a vault, enforce that the previous vault linked by the user belongs to the same denomination as the currently modified vault.

Status

This finding has been addressed by enforcing that vaults in any sort of relationship should have the same denomination (commit id `a9c2bfbbb441f353d77c6733ddbc610e0dc66f42` in the branch `audit-fixes`).

[A9] Vaults can be configured so that

`opening_col_rate` < `min_vault_col`

Severity: Low

Difficulty: Medium

Recommended Action: Fix Code

Addressed by client

Description

If a vault ever has a collateralization rate that is less than the minimum allowed value stored in the contract (`min_col_rate`), then the vault is able to be liquidated. There is a check to make sure that a vault does not have a collateralization rate less than the minimum opening collateralization rate (`opening_col_rate`) from a call to the function `new_vault` or `increase_debt` . However, it is possible that the protocol could be configured so that `opening_col_rate` can be lesser than `min_vault_col` , this would mean that even though the checks would pass, the vault would be able to be liquidated immediately after a call to these functions.

Recommendation

Add a check enforcing that `opening_col_rate` should be greater than `min_col_rate` whenever either one is written.

Status

This finding has been addressed by adding the appropriate guard in the `set_vault_conditions` function (commit id `0b15c2d3ad669938f89320adaf5acc84877b8d2a` in the branch `audit-fixes`).

[A10] No sanitation of `fee` for `vaults` or `stable-liquidity-pool`

Severity: Low

Difficulty: Medium

Recommended Action: Fix Code

Addressed by client

Description

The Vaults and Stable Liquidity Pool contracts charge a fee for users in most of the operations that move assets within or to and from the contracts. This fee is charged as a percentage of the funds that are being handled, and that percentage is stored in `fee` and `fee_percentage` fields of `CoreState`, for Vaults and Stable Liquidity Pool respectively. However there is no input validation on the setting of these fields, and it is possible for both fields to be set to 100% or greater.

Having the charged fee at or above 100% will make the protocol unusable. If made 100%, the usage of the contracts becomes financially unfeasible for users having in mind the protocol's intended business logic. Furthermore, if made above 100%, the fee will cause the contracts to panic when attempting to deduce the protocol share from the amount being manipulated by the user due to an integer underflow.

Recommendation

Add input validation to ensure the fees are set to an appropriate range.

Status

This finding has been addressed by the client by setting a maximum fee of 1% for Vaults and 5% for Stable Liquidity Pools (commit ids `6e4c41dd209c0e276ec0fc5add26678b5dcf3402` and `1499c90360cbc171f9ad3c584284a421fd1ae34e` on the branch `audit-fixes`).



Informative Findings

The findings presented in this section do not necessarily represent any flaw in the code itself. However, they indicate areas where the code may need external support or deviate from best practices. We have also included information on potential code size reductions and remarks on the operational perspective of the contract.



[B1] Best Practices and Notable Particularities

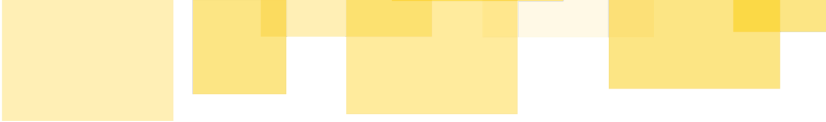
Severity: Informative

Partially addressed by client

Description

Here are some notes on the protocol particularities, comments, and suggestions to improve the code or the business logic of the protocol in a best-practice sense. They do not in themselves present issues to the audited protocol but are advised to either be aware of or to be followed when possible, and may explain minor unexpected behaviors on the deployed project.

1. While the Vault contract uses the generation time of ledgers for calculating time windows, Stable Liquidity Pool contracts use a more accurate approach of using timestamps instead;
2. The business logic for the Vault contracts considers an average ledger time of 5 seconds. This value is used for calculating the number of ledgers in a range of days for maintaining the contract's storage. The ledger time is not fixed and may vary, and a brief observation of the [Stellar.org Dashboard](#) shows us that block times can have an average of up to 6 seconds in a range of 200 blocks. Although not an issue, it may hinder the accuracy of the time ranges handled in the protocol;
3. The `init` function in the `Vault` contract has a `stable_issuer` address given to it as a parameter. This address is never used;
4. The vault and stable liquidity pool contracts can be instantly upgraded. This may be a source of trust issues with users;
5. There are many instances in the system of unchecked arithmetic operations. While this contract has `overflow-checks` set to `true` in the `Cargo.toml`, the best practice is to use the `checked_<OPERATION>` versions of the arithmetic operations, for example `checked_add` instead of `+`. This also allows for more targeted error messaging in handling the returned overflow case;
6. Functions responsible for user-protocol interactions often require information as parameters that could be inferred within the business logic of the contracts. This expands the surface of possible exploits based on the different possibilities of inputs that a user can provide (as well exemplified by findings [A3](#) and [A8](#));

- 
7. Ceiling and flooring division operations in the contracts are performed using the implementation provided in the `num-integer` library, instead of the native operations provided within the used types.
-

Recommendations

For each of the topics elaborated above, we recommend implementing the following approaches into the protocol's contracts:

1. Maintain a pattern of the approaches used for calculating time windows throughout the protocol's contracts, modifying the Vault contract to use timestamps instead of ledger numbers;
 2. If the previous recommendation is not followed, in the Vault contract, allow ledger generation time to be modified by administrators of the protocol to better reflect the throughput of the blockchain in the future;
 3. Remove the `stable_issuer` from the `init` function in the current version of the Vault contract;
 4. As a basis for establishing trust with users of the protocol, features such as upgrading a contract should be handled with measures for protection, such as having the protocol manager submit the new contract hash to the contract and having to wait a set amount of time before actually upgrading to it. This allows users to evaluate possible changes to the protocol, enhancing trust;
 5. Whenever possible, use checked arithmetic operations to enhance code reliability and improve the users' comprehension when interacting with failed operations;
 6. Reduce the surface of interaction between the user and the protocol by removing unnecessary parameters from functions;
 7. If an implementation of an operation is available natively, avoid using a third-party library to perform the same operation.
-

Status

This finding has been partially addressed by the client. Considering the topics mentioned above, the client has:

- Removed the `num-integer` dependency from the contracts.



[B2] User Can Deposit and Lock Deposits of Zero Tokens

Severity: Informative

Addressed by client

Description

When creating a deposit, no validations are performed over the value that the user is providing to the protocol. This means that a user can deposit 0 assets and, by doing so, will have a `Deposit` storage instance created, even if the user has no shares to redeem.

Furthermore, the only requirement for a user to lock his deposit and to participate in the distribution of protocol governance tokens is that the storage instance of the user deposit exists. The consequence of this is deposits of zero tokens can be locked.

By themselves, these two particularities are harmless, as the user will not receive any benefits from depositing and locking 0 assets, as shares will not be issued, and the protocol governance token rewards are calculated based on the individual number of shares issued to each user.

Furthermore, the fact that no validations are performed in these two scenarios can lead to unpredicted behaviors on the protocol, as demonstrated by aggravating finding [A1](#).

Recommendations

Require a minimum deposit amount to prevent users from having a `Deposit` storage instance with no deposit. Furthermore, validate that a `Deposit` instance either has shares issued or a deposited amount tied to it to allow locking it.

Status

This finding has been addressed by enforcing a minimum deposit amount (commit id `75f88ea6588973a386deb26d64b31ca6288efcdc` in the branch `audit-fixes`).



[B3] It is Possible to Swap Assets In Liquidity Pools Without Paying Fees

Severity: Informative

Not addressed by client

Description

Whenever depositing or withdrawing funds from a FxDAO's stable liquidity pools, no fees are charged to the user in any of the two operations. Initially, this would open up the liquidity pools to a vulnerability of allowing users to swap assets without having to pay fees, which would be done by depositing and withdrawing the desired amounts consecutively.

FxDAO's liquidity pools circumvent this issue by enforcing that a user has to wait roughly 48 hours to be able to withdraw any amount of his deposited liquidity after a deposit. This prevents the protocol from being a part of atomic operations where, for instance, a user exploits an arbitrage opportunity using flash loans and profits from not paying protocol fees. Still, if the user is not in a hurry to perform the swap, given that the protocol exchange rate for the tokens is always 1-to-1, he can still perform it in the course of two days without paying fees.

Status

This finding has been acknowledged by the client.



[B4] Dependency on Governance Token Trust

Severity: Informative

Not addressed by client

Description

The Stable Liquidity Pool contract rewards users with a share of governance tokens for locking their assets in the liquidity pools. This share is proportional to the length of time their assets are locked. However, the governance token is not minted by the contract. Instead, the minting, distribution, burning, and total supply are handled externally.

While the protocol owners have a system for all of these factors, since nothing is enforced by the Stable Liquidity Pool contract itself, this presents a layer of trust that the handling of the governance token is done appropriately and is resistant to outside exploitation.

Status

This finding has been acknowledged by the client.



[B5] Contracts May Trigger Misleading Panic Messages

Severity: Informative

Not addressed by client

Description

For both the Vaults and Stable Liquidity Pool smart contracts, different reasons for errors may trigger the same error code/message. The situations are described below:

- **Vaults:**

Whenever creating a new vault or increasing a vault's debt, a conditional is used to validate if the protocol is in panic mode or if the currency exchange rate is outdated. If one of the two elements in this conditional is true, the contract handles the situation by panicking with a single error code.

Considering the enumerator with error codes documented for this protocol, the error code triggered in this conditional is named `PanicModeEnabled`, which, in a scenario where the currency is outdated, could mislead a user to believe that the protocol is in panic mode.

- **Stable Liquidity Pool:**

When attempting to unlock the deposit on a stable liquidity pool, the attempt may fail if the user doesn't have a deposit recorded in the contract, or if his deposit is not locked. The error code is named `NotLockedDeposit` and, while it reflects the fact that the user is not in a position to lock a deposit, the error may be triggered by two different causes.

Recommendation

For the vault smart contract, have different checks to evaluate if the protocol is in panic mode and if the currency exchange rate is outdated. Also, introduce a new error code for failure to operate due to an outdated currency exchange rate.

Similarly, for the stable liquidity pool, trigger an error when the user doesn't have a deposit registered, and another when the user has a deposit but hasn't locked it yet.



Status

This finding has been acknowledged by the client.



[B6] The Vaults Protocol Does Not Allow the Removal of Currencies

Severity: Informative

Not addressed by client

Description

For the Vaults smart contract, as previously elaborated, the process of providing backing collateral to an asset and issuing debt to a user happens in the form of creating a vault. To create a vault, the protocol first has to instantiate a currency, which aims to specify the name or Symbol of an asset that users can back by providing collateral, while also specifying a token address used as the representation of the issued debt, or the loan taken by the user.

Currencies can be created and toggled active/inactive, and the second operation is responsible for defining when operations over the vaults of this currency can be performed. Still, there are no mechanisms that allow currencies to be removed from the contract, which may become problematic if a currency is created with incorrect information, or if for any reason the denomination represented by that currency becomes corrupted, and has no more use within the denominations now stored in the Vaults contract.

Recommendation

Introduce a function to allow the removal of currencies from the Vaults contract.

Status

This finding has been acknowledged by the client.



[B7] Only One Oracle is Used As Reference for Exchange Rates and No Validations Over The Fetched Values Are Performed

Severity: Informative

Not addressed by client

Description

In the Vaults contract, when creating a loan against a backed asset, the exchange rate of this asset to the collateral token is used in operations to define the health of users' vaults. the FxDAO Vaults protocol uses a single oracle as the source of its exchange rates, and no operations over the values fetched by that oracle are performed other than validations on how old the data is.

This creates a trust dependency between the Vaults contract and this oracle, where a single ledger in which the oracle contract may feed incorrect data has the potential to cause irreparable damage to the economic health of this protocol.

Recommendation

Following the concept that smart contracts should be as self-sufficient and reliable as possible, introduce mechanisms to prevent sudden spikes in value from damaging the protocol, such as circuit breakers and using moving averages instead of the raw exchange rate.

Furthermore, have as many sources of exchange rates as possible to avoid relying on a single oracle contract.

Status

This finding has been acknowledged by the client.



[B8] Stable Liquidity Pools have a Constant Exchange Rate

Severity: Informative

Not addressed by client

Description

As mentioned in the [Platform Logic and Features Description](#), the exchange rate of assets in Stable Liquidity Pools is constant at all times. While this does not necessarily reflect an issue in the protocol, it is known that stablecoins can depeg for different reasons, which would mean that, in the case of a stablecoin depegging event, tokens of potentially different values would be traded in a fixed exchange rate.

Given that the Stable Liquidity Pool contract (by itself) has no internal mechanisms to shut down operations if a variation in the price of a stablecoin is detected, the protocol is subject to the possibility of being used in exploits or being a central part in massive withdrawal events (popularly known as bank runs) in case of unexpected stablecoin price variations.

Status

This finding has been acknowledged by the client.



[B9] `protocol_manager` and `Admin` can be the same address

Severity: Informative

Not addressed by client

Description

The privileged roles of `admin` and `protocol_manager` for both `vaults` and `stable-liquidity-pool` contracts are intended to be governance, and a trusted FxDAO member. However, while this is the intention, it is entirely possible that the same address could be accidentally or intentionally entered for both roles.

It should also be noted that there are no setter functions for `admin` and `protocol_manager` in the Stable Liquidity Pool contract. In the event that they should be changed, there is no way to do this without these functions.

Recommendation

Add a logic to ensure that both roles are distinct, and add setter functions to the Stable Liquidity Pool contract for both roles.

Status

This finding has been acknowledged by the client.



[B10] Rounding on Division Allow Users to Withdraw Dust From Liquidity Pools

Severity: Informative

Addressed by client

Description

As mentioned in the [Platform Logic and Features Description](#) section, specifically on the calculation of new share prices when performing swaps on the Stable Liquidity Pool contract, the new value for the price of a share is obtained as the result of a floored division. One of the consequences of having this division operation rounded down is the number of shares issued for deposits will be greater than if the share price division were rounded up.

After analyzing the above conditions under a concrete scenario, it was discovered that, if rounding down the calculation of share prices, a user could withdraw a slightly higher number of assets than he originally deposited without a swap occurring between his deposit and withdrawal.

Scenario

The scenario consists of the following:

1. The pool is populated by one or more users providing a total of 1234_5678987 tokens in liquidity;
 2. For the simplicity of the example, we will assume that no swaps in this pool were performed up to this point, meaning that the share price is 1_0000000;
 3. At this moment, 10 swaps of the total value of the pool are performed. The share price will be different depending on if we use ceil or floor for the division: 98_5123550 shares if the division is floored, and 98_5122580 if the division is ceiled;
 4. After the swaps, a user joins the pool with 100_0000000 tokens, getting a different number of shares issued for each scenario;
 5. The user withdraws his liquidity, earning his tokens back. No swaps were performed between the user's deposit and withdrawal. If using the floored division, the user withdraws 100_0000506 tokens, even though no pool profit happened for the duration of his deposit.
-



Recommendation

When calculating the share price, use `ceil` divisions instead of `floor` .

Status

This finding has been addressed by the client (commit id `0b15c2d3ad669938f89320adaf5acc84877b8d2a` on the branch `audit-fixes`).