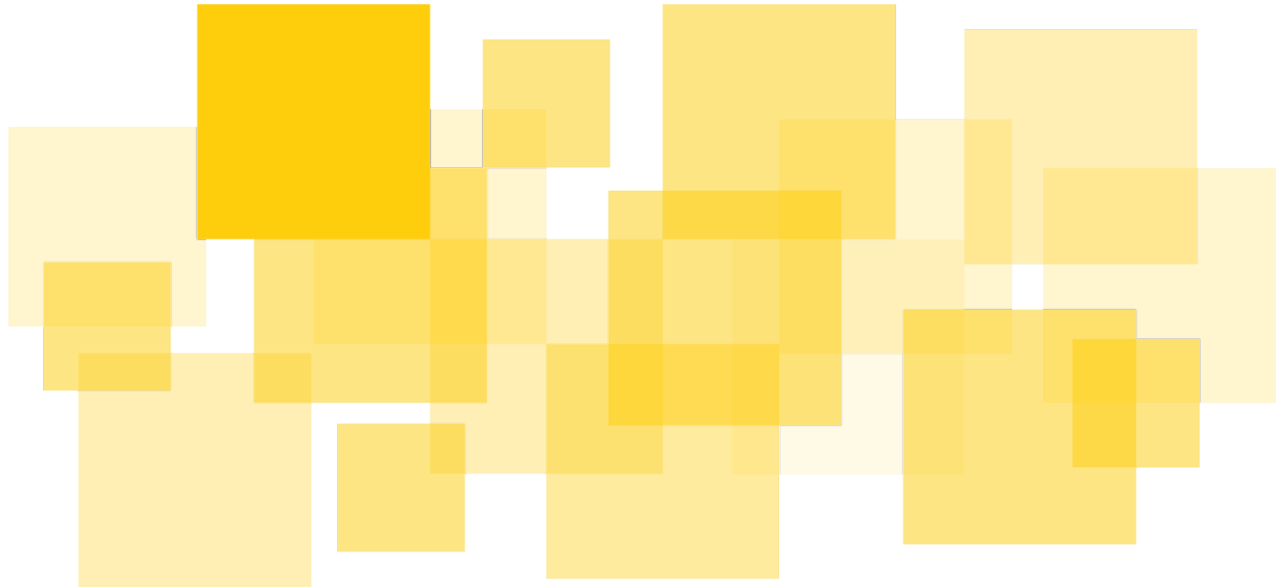


Security Audit Report

HydraDX - Stableswap Polkadot

Delivered: July 24, 2023



Prepared for HydraDX by





Table of Contents

- Disclaimer
- Executive Summary
- Goal
- Scope
- Methodology
- Platform Logic and Features Description
- Differences From Canonical Stableswap Invariant
- Findings
 - [A1] Lack of validations when updating the stableswap's amplification factor
 - [A2] Single and empty asset list is permissible as an argument to `create_pool`
 - [A3] `add_liquidity` does not check for duplicate assets in `assets` argument
 - [A4] The asset balance of a pool can drop below the existential deposit
 - [A5] Filtering non-zero assets in `calculate_d` and `calculate_y` breaks non-degeneracy assumption
 - [A6] Withdrawal of assets is susceptible to slippage
 - [A7] Withdrawing all share assets triggers errors
 - [A8] It is possible to create pools with, or update pools to have, identical configurations
 - [A9] Unexpected interactions and configuration of pools can be achieved by directly sending assets to them
 - [A10] Possibility of sandwich attacks by consecutive modifications on the amplification factor
- Informative Findings
 - [B1] Best practices
 - [B2] Fees are unbounded
 - [B3] `set_asset_tradable_state` does not check if the `asset_id` input is in the pool
 - [B4] The absence of a token in a pool may lead to misleading error messaging
 - [B5] Presence of identified potential vulnerabilities in dependencies of crates
 - [B6] Deviation from Curve stableswap invariant implementation and calculation strategy



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

HydraDX engaged [Runtime Verification Inc.](#) to conduct a security audit of their pallet's 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 3.5 calendar weeks (April 31, 2023 through June 23, 2023) and focused on analyzing the security of the source code of HydraDX's stableswap system, which enables users to deposit and withdraw assets into 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:

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

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

- Best practices: [B1](#), [B2](#), [B3](#), [B4](#)
- Code optimization-related particularities: [B1](#)
- Potentially compromised dependencies: [B5](#)
- General observations about the protocol math: [B6](#)

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 HydraDX's system based on the provided documentation;
2. Review the low-level implementation of the system for the individual Rust modules (stableswap core and math modules);
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 repositories of two functional modules provided by the client. These projects are:

1. HydraDX-node ([Commit 091bc9d084139bfc1d7899534d311f7f2f46dbdb](#)): this repository contains all modules included in the HydraDX's node. From it, two modules were audited:
 - Stableswap core (`pallets/stableswap/src`): implements the interactions between users and stableswap pools, as well as manages these pools. The features here include, but are not limited to: adding liquidity, removing liquidity, buying and selling assets, creating and updating pools;
 - Math module (`math/src/stableswap`): implements the mathematic operations performed by the protocol in order to obtain values used by it;

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 a deployed contract.

Second, we thoroughly reviewed the pallet 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 of size and types of variables of the utilized modules, 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 HydraDX's stableswap logic;
- Created formal models of code snippets responsible for the calculations in core components of the protocol to mathematically prove its safety.

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

Tools capable of identifying dependency-related issues such as cargo-audit and cargo-geiger have been used to analyze possible security issues in crates referenced in this code.

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.

Additionally, given the nascent Polkadot 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

HydraDX's stableswap protocol is a platform built over the Polkadot network and implemented at the L1 level that allows users to swap and also stake their stablecoins to obtain liquidity pool tokens, granting the users the benefit of receiving a portion of the fees of operations performed on the protocol's liquidity pools. These liquidity pools follow the stableswap invariant for calculating constants responsible for keeping the values of the pools, with some adjustments.

The canonical methodology aims to merge the qualities of the constant price market maker invariant ($x + y = \text{const}$) and the constant product market maker invariant ($x * y = \text{const}$)

The integration of these two equations, with the addition of an Amplification Factor A such that $A \in \mathbb{Q} \geq 0$ results in the following invariant equation:

$$An^n * \sum X_i + D = D * An^n + \frac{D^{n+1}}{(n^n * \prod X_i)} \quad (1)$$

Where we highlight that:

1. X_i represents the asset balance of the pool, where a pool can hold between 2 and 5 assets in the HydraDX stableswap protocol;
2. n represents the number of assets in a stableswap pool;
3. D is not given, i.e., we must solve the above equation for D ;
4. the equation's behavior becomes more like a constant product invariant when A is small and more like a constant sum invariant when A is large.

Furthermore, for our analysis, we note some invariants that must be maintained by a stableswap protocol that has no empty balances to ensure that calculations performed in operations do not unjustly benefit (or happen to the detriment of) a user or the protocol:

- $0 < A$
- $\forall x_i \in X. 0 < x_i$
- $0 < D$

The assertions above constitute what we refer to as the **non-degeneracy** of a protocol's health.

Given the typing limitations faced when calculating the constant D of the automated market maker, the Newton-Raphson method is used for obtaining an approximated value. This is a popular and battle-tested solution for solving the stableswap invariant, as it was implemented by Curve protocol and subsequently used in a number of protocols with shared purposes.

Deposits, withdrawals, buy and sell operations all rely on the calculation of D to return the number of tokens that will be provided to a user.

Differences From Canonical Stableswap Invariant

HydraDX stableswap protocol differs from the [original Curve proposed specification](#) of the stableswap invariant as demonstrated below:

Calculation of D is performed iteratively following the Newton-Raphson method for the next approximation of D_{next} using the previous approximation D_{prev} until convergence occurs between D_{next} and D_{prev} within a desired precision. When convergence is achieved, $D = D_{next}$. For equation (1), this is represented by:

$$D_{next} = \frac{D_{prev} * (n * D_p + A n^n * S)}{(n+1) * D_p + D_{prev} * (A n^n - 1)} \quad (2)$$

where

$$D_p = \frac{(D_{prev})^{n+1}}{n^n * \prod X_i}$$

However, in order to avoid integer overflow for the numerator and denominator at the cost of precision, D_p is calculated by:

$$D_p = D_{prev} * \frac{D_{prev}}{n * x_0} * \frac{D_{prev}}{n * x_1} * \dots * \frac{D_{prev}}{n * x_{n-1}} \quad (3)$$

HydraDX stableswap protocol has deviated from this approach and introduced the addition of $+2$ in the calculation of (2):

$$D_{next} = \frac{D_{prev} * (n * D_p + A n^n * S)}{(n+1) * D_p + D_{prev} * (A n^n - 1)} + 2 \quad (4)$$

The protocol also requires the calculation of x_j , where $x_j \in X$ and the balance of x_j is unknown, and $\forall x_i \in X. x_i \neq x_j \implies$ balance of x_i is known. To match the naming convention of the code, we will now refer to x_j as Y . Similarly to D , calculation of Y is performed iteratively *w.r.t.* Y_{next} and Y_{prev} , and once convergence is achieved between the two, $Y = Y_{next}$. Y is calculated by:

$$Y_{next} = \frac{(Y_{prev})^2 + c}{2Y_{prev} + b - D} \quad (5)$$

where

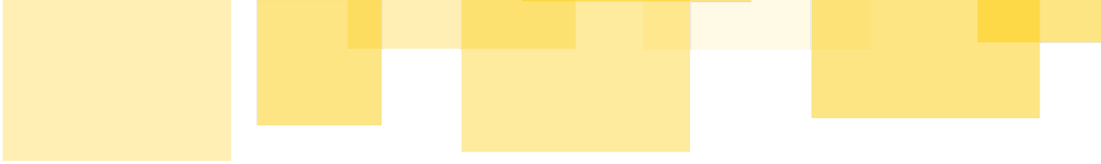
$$b = \sum X_{i \neq j} + \frac{D}{A n^n} \quad (6)$$

$$c = \frac{D^{n+1}}{A n^n * n^n * \prod X_{i \neq j}} \quad (7)$$

and similarly to the deviation in the calculation of D , HydraDX have additionally deviated from the original invariant by adding $+2$ to the calculation of (5):

$$Y_{next} = \frac{(Y_{prev})^2 + c}{2Y_{prev} + b - D} + 2 \quad (8)$$

It can be seen in the above equations that the calculation of (4) and (8) is recursive, therefore the effect of the addition of $+2$ for each equation is also recursive. This means that the range of



values differing between the original Curve and HydraDX implementations is larger than a linear raise of $+2$, and may have subtle effects on the protocol.

Part of our analysis of the HydraDX implementation included a comparison of the results of the internal and external functions between the two. Our analysis and testing indicate that the changes have the following effects:

let D be the result of (2) after convergence,

let D' be the result of (4) after convergence,

let Y be the result of (5) after convergence,

let Y' be the result of (8) after convergence,

let $F(e)$ be that equation e errored by overflow, underflow, or divide zero, where $e \in \{(2), (4), (5), (8)\}$

- $D \leq D'$
- $Y \leq Y'$
- $2 \leq D'$
- $2 \leq Y'$
- $F(4) \implies F(2)$
- $F(8) \implies F(5)$
- $\neg(F(2) \implies F(4))$
- $\neg(F(8) \implies F(5))$
- After sufficient initial liquidity is provided $\forall x_i \in X. 2 \leq \text{balance}(x_i)$

These properties have been considered and are believed to hold for all values provided to the functions. They were also confirmed with property tests in Rust.

These changes and their outcomes are not indicative of the complete set of differences between the two protocols, and while these changes and outcomes are true and accurate to the best of our knowledge, they have not been formally verified.



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] Lack of validations when updating the stableswap's amplification factor

Severity: Low

Difficulty: Medium

Recommended Action: Fix Code

Addressed by client

Description

As described in [the Platform and Features Description section](#), the amplification factor is used to define the proximity of the behavior of the stableswap invariant to either the constant product or constant price invariants, playing an essential role on the calculation of issued share assets and the exchanged amount of assets on trades.

In HydraDX's stableswap protocol, the amplification factor is not explicitly restricted as non-zero, breaking non-degeneracy assumptions. Represented as `AmplificationRange` in storage, this value is instantiated at runtime as a range of `u16`, and is defined through the protocol's governance.

Although unlikely, it is possible that by exploiting possible issues in the governance, or through griefing attacks performed by a majority of hypothetically malicious users, the amplification factor can be set to, for instance, 0. This would effectively break the invariant in which this protocol is built upon.

Recommendation

Add a specific check ensuring that the amplification factor is a non-zero value at the `create_pool` and `update_pool` functions, or restrict `AmplificationRange` in storage to the range provided by the type `NonZeroU16`. This would also lead to the `PoolInfo` field `amplification` being changed from type `u16` to `NonZeroU16`.

Status

The client has addressed the finding by following the recommendation provided above.



[A2] Single and empty asset list is permissible as an argument to `create_pool`

Severity: Low

Difficulty: High

Recommended Action: Fix Code

Addressed by client

Description

When creating a stableswap pool, the protocol receives the following parameters for the `create_pool` function: the origin of the function call (`origin`), the id of the liquidity share asset (`share_asset`), the list of assets to be held by this pool (`assets`), the amplification factor (`amplification`) for this pool and finally its trade and withdrawal fees (`trade_fee` , `withdraw_fee` , respectively).

Considering these parameters, there is no validation on the `create_pool` function to ensure that the vector representing the list of assets held by the pool has at least two assets. A pool with zero, or one asset is not usable to swap assets, and can only increase the options for potential attack vectors.

Recommendation

Add a check function in the `PoolInfo` entity named `is_valid` which checks that there are at least 2 assets in the array of assets.

Status

The client has addressed the finding by following the recommendation provided above.

[A3] `add_liquidity` does not check for duplicate assets in `assets` argument

Severity: Medium

Difficulty: Medium

Recommended Action: Fix Code

Addressed by client

Description

When adding liquidity to one of the pools in the protocol, a user can call the `add_liquidity` function providing the following parameters: the origin of the call (`origin`), the id of the pool in which liquidity is being added (`pool_id`), and a list of assets and amounts to be deposited (`assets`).

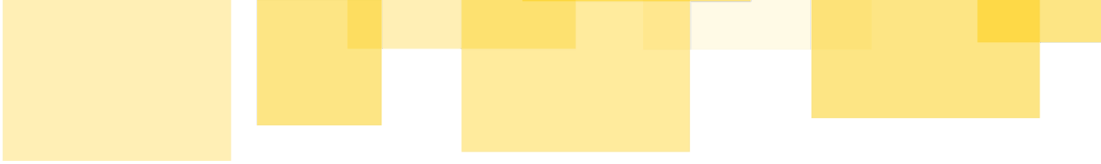
While processing the amounts to be deposited in the pool a map of asset-to-amount-deposited is built. The code excerpt below shows the creation of this map, named `added_assets` :

```
let mut added_assets = BTreeMap::<T::AssetId, Balance>::new();
for asset in assets.iter() {
    ensure!(
        Self::is_asset_allowed(pool_id, asset.asset_id, Tradability::ADD_LIQUIDITY),
        Error::<T>::NotAllowed
    );
    ensure!(
        asset.amount >= T::MinTradingLimit::get(),
        Error::<T>::InsufficientTradingAmount
    );
    ensure!(
        T::Currency::free_balance(asset.asset_id, who) >= asset.amount,
        Error::<T>::InsufficientBalance
    );
    ensure!(pool.find_asset(asset.asset_id).is_some(), Error::<T>::AssetNotInPool);
    added_assets.insert(asset.asset_id, asset.amount);
}
```

Notice that the map being discussed here is a `BTreeMap` , and if trying to insert a key that already exists in that map, the value for that key will be updated. No error will be thrown, and no two identical keys will exist in that map.

After being built, `added_assets` will be used to calculate the number of shares issued to a user when performing a deposit to the pool. In contrast, once the shares are calculated, the protocol proceeds to transfer the deposited assets to the pool and issue the share tokens to the user, but it uses the `assets` vector provided as a parameter for the `add_liquidity` function call to specify the transfers that will be performed from the user to the pool.

This means that if a user, either by mistake or misconfiguration at the end of the user or a third party handling the interaction of the user with the protocol, submits a deposit operation with repeated tokens in the `assets` vector, only the last of the repeated instances of this vector will



be considered in the calculation of shares for that user, but for each asset and amount in the `assets` vector there will be a transfer from the user to the protocol.

Scenario

For better comprehension of the finding, consider the steps of the following scenario:

1. A pool A of USDC, USDT, and DAI exists and has liquidity on it;
 2. User Bob attempts to add liquidity to pool A but mistakenly informs that will deposit 50 USDC, 10 USDC, and 30 DAI in the `assets` parameter;
 3. The protocol creates the `added_assets` map and iterates over the `assets` vector, performing validations and adding each individual token to the map;
 4. As USDC is repeated in the `assets` vector, only the last value for that key will remain in the map, meaning that `added_assets` contains two instances:

```
{ 'USDC': 10, 'DAI': 30 } ;
```
 5. The number of shares issued to that user will consider only a total of 10 USDC and 30 DAI deposited into the protocol;
 6. 60 USDC and 30 DAI will be transferred from the user to the protocol, where the first 50 USDC transfer will be handled as a donation.
-

Recommendation

`BTreeMap` function `insert` returns a value depending if the key already existed in the key set, [documentation](#). This value should be captured and checked that it is always `None` so that no duplicate assets are added.

Status

The client has addressed the finding by following the recommendation provided above.



[A4] The asset balance of a pool can drop below the existential deposit

Severity: High

Difficulty: Medium

Recommended Action: Fix Code

Addressed by client

Description

To avoid issues of blockchain state bloating, Substrate-based chains adopt a mechanic named Existential Deposit (ED). ED works by ensuring that only accounts with a minimum amount of an asset are stored in the blockchain state, reducing the size of data handled by the blockchain node and enhancing performance. It comes with a consequence, [as mentioned by Polkadot Support](#):

If an account drops below the ED, the account is reaped ("deactivated"), and any remaining funds are destroyed. The address can be reactivated with a new deposit larger than the existential deposit at any time. This will not restore the destroyed funds.

On the implementation of the blockchain mechanics for HydraDX's node, the transfer function is configured with the flag `AllowDeath` enabled. When transferring tokens from a pool account to user accounts, there is no check to prevent the asset balance of the pool to drop below the existential deposit. If that happens, the pool account will be reaped and the asset balance will be destroyed. It will break the invariant assumption ensuring that the asset balances must be greater than zero after the provision of initial liquidity.

Recommendation

The Hydradx team should carefully add all the pool accounts to the `DustRemovalWhitelist` or ensure that the pools' balances will always remain above the ED threshold.

Status

The client has addressed the finding by following the recommendation provided above.

[A5] Filtering non-zero assets in `calculate_d` and `calculate_y` breaks non-degeneracy assumption

Severity: Medium

Difficulty: High

Recommended Action: Fix Code

Addressed by client

Description

One of the core concepts of non-degeneracy of the stableswap invariant requires that, after initial liquidity is added, all assets in the pool's asset list must have a non-zero balance.

Both `calculate_y` and `calculate_d` functions, which are used to calculate the number of tokens to be traded by the protocol, disregard this requirement by filtering out zero balance assets from the provided asset list, as seen in the sample code below:

```
let mut xp_hp: Vec<U256> = xp.iter().filter(|v| !(*v).is_zero()).map(|v| to_u256!(*v)).collect();
```

Where `xp` represents a vector of asset balances of the pool. `xp` is only valid if either no assets have zero balance, or all assets have zero balance (initial state of the pool).

Given that there are no checks ensuring that the protocol balances never go below the existential deposit for the handled tokens ([further elaborated in finding A4](#)), and that the enforcement of the whitelisting of all pools in order to prevent their balances from being reaped by the blockchain mechanics becomes a blockchain-implementation-related issue, this constitutes a threat to the health of the protocol.

Recommendation

On the `calculate_y` and `calculate_d` functions, add a check to ensure that either all assets are zero balance, or no assets are zero balance.

Status

The client has addressed the finding by validating that, after filtering out the assets with zero balance, the list length after the filtering remains the same.

[A6] Withdrawal of assets is susceptible to slippage

Severity: Medium

Difficulty: Medium

Recommended Action: Fix Design

Partially addressed by client

Description

The only method of withdrawing assets available to a user is `remove_liquidity_one_asset`, this burns the users `liquidity tokens` in exchange for a chosen asset. This means that liquidity provided with a large number of shares is able susceptible to slippage when withdrawing tokens. Consider the following scenario:

1. A 3-token pool is created with tokens (a, b, c)
2. A liquidity provider LP deposits n amount of each token a, b, c into the pool, receiving $3n$ liquidity shares (NOTE: all three assets MUST be provided for the initial call to `add_liquidity`)

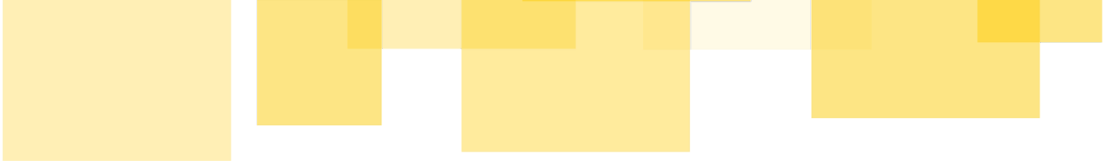
Now consider that for step 3 a LP attempts to divest all of their $3n$ liquidity shares for asset a (arbitrarily chosen). Consider different strategies available to achieve this:

- LP calls `withdraw_share_one_asset` with argument `share_amount == 3n`. If `amount_out` is the quantity of asset a transferred to LP after burning the liquidity tokens, they would experience extreme slippage as `amount_out` $< n$. There is no parameter for `min_amount_in`, so LP is completely unprotected from slippage.
- LP makes 3 calls to `withdraw_share_one_asset` each with `share_amount == n`, however changing asset from a , to b , to c . This method is the best option available for an LP to recover funds.
- LP attempts to avoid slippage by unbalancing the pool first calling `sell` twice with arguments `asset_in == a; asset_out == b`; then `asset_in == a; asset_out == c` each with `amount_in == n'` where $n' < n$ to avoid slippage. Then LP attempts to divest their $3n$ liquidity tokens via the first method listed. However $n < amount_out < 3n$ as the liquidity will never be transferred entirely to asset a in the pool. Also, LP has now paid `withdraw_fee` and `transfer_fee` twice, and they must have the funds available to unbalance the pool.

All three options are sub-optimal from a user's perspective to varying degrees.

Recommendation

Two suggestions are provided to address the topic elaborated above:

- 
- Implement a function to remove liquidity from multiple assets in the pool, consider Curve finance `remove_liquidity` and `remove_liquidity_imbalance` ;
 - Add parameter `min_amount_out` to function `remove_liquidity_one_asset` so that users can protect themselves from slippage.
-

Status

The client partially addressed the finding by implementing a minimum amount received when removing liquidity from the pool. Issues with slippage can still happen, as the individual and sequential withdrawals of assets from a pool imply possible slippage issues for users.

[A7] Withdrawing all share assets triggers errors

Severity: Low

Difficulty: Medium

Recommended Action: Fix Code

Not addressed by client

Description

It is possible to attempt a withdrawal with all share assets that have been issued, with the user withdrawing being the sole liquidity provider for the pool

(`current_share_balance == share_amount`). This is also explicit in the check within the `remove_liquidity_one_asset` function:

```
ensure!(  
    current_share_balance == share_amount  
    || current_share_balance.saturating_sub(share_amount) >= T::MinPoolLiquidity::get(),  
    Error::::InsufficientShareBalance  
);
```

When a user holds the remaining liquidity tokens, an attempt to divest will trigger an overflow error in `calculate_y`.

This happens when calculating the number of remaining reserves after the withdrawal, precisely in the operation in line 172 of the `math/src/stableswap/math.rs` module

(`reserve.checked_mul(d1)?.checked_div(d_hp)?.checked_sub(y_hp)?`). In this scenario, if the user is trying to withdraw using all share assets in issuance,

`reserve.checked_mul(d1)?.checked_div(d_hp)` yields 0, and `y_hp` yields 2 due to the addition of two when calculating y (diverging from Curve's original implementation).

In addition, the user should not be able to withdraw all share assets in the

`remove_liquidity_one_asset`, as it implies that there will be no issuance of share assets while only one asset of the pool has been withdrawn.

Recommendation

Do not allow users to withdraw using all share assets issued using the


`remove_liquidity_one_asset`, implementing a strategy to withdraw portions from assets in the pool as proposed in finding [A6](#).

Another approach for handling this issue is modifying the `checked_sub` operation in the mentioned calculation for a `saturating_sub` to avoid running into overflow issues. Still, this could introduce new possibilities for reducing a balance of an individual asset to 0, and we do not recommend its implementation without the thorough analysis and testing of its safety, as well as proper validations to maintain the non-degeneracy assertions for this protocol.



Status

This finding has been acknowledged by the client.



[A8] It is possible to create pools with, or update pools to have, identical configurations

Severity: Low

Difficulty: High

Recommended Action: Fix Code

Not addressed by client

Description

When creating pools, it is necessary to provide the id of the share asset issued by the pool, the list of assets to be added as liquidity, the amplification factor, and the trade and withdrawal fees. When updating the pools, all the same parameters are provided, with the exception of the liquidity assets list.

Although validations over the assets held and issued by the pool are performed, the code that handles these functionalities do not validate the existence of a pool with the same configuration as the one being created/updated. This means that two pools with the same settings (with the exception of the issued shared asset id) can exist.

The existence of two identical pools does not imply the existence of attack vectors for malicious users, but does imply the liquidity being split between multiple pools. Given that these pools have no mechanisms built on top of them (for instance Curve's liquidity boosting by locking share assets), there is no motivation to split the revenue of liquidity providers in different pools. Furthermore, this also raises the price impact of users' trades.

Recommendation

Validate if a pool being created or updated has the same configuration as the ones that already exist.

Status

The client has acknowledged the finding.



[A9] Unexpected interactions and configuration of pools can be achieved by directly sending assets to them

Severity: Low

Difficulty: Medium

Recommended Action: Fix Design

Not addressed by client

Description

Explicitly enforced in the code of HydraDX's stableswap logic is the requirement that the first deposit must be made by sending an amount of every token that the pool is supposed to hold. This complies with the non-degeneracy assertions by making sure that either the pool has no tokens, or at least some amount (greater than zero) is held for each token that the pool is supposed to hold.

What this does not consider is the possibility of directly sending tokens to the pool, breaking part of the non-degeneracy assumptions that are used to keep the pool healthy. This also configures a scenario where liquidity tokens are in the pool but no amount issued share asset exists.

To the best of our knowledge, no major exploit can be achieved by sending tokens to a recently created pool other than twisting the exchange rate of tokens in that pool, where the amount sent will be considered a donation to the first liquidity provider.

Moreover, in a scenario where assets are sent to a pool after it received its initial deposit, manipulation of the exchange rate can still be achieved by directly sending tokens to the pool.

Recommendation

Avoid the consequences of unexpected interactions with the pools by blocking the direct transfer of assets to them or, regarding the initial liquidity requirement for pool creation, have the creation of the pool and addition of the initial liquidity within the same transaction group.

Status

The client has acknowledged the finding.



[A10] Possibility of sandwich attacks by consecutive modifications on the amplification factor

Severity: High

Difficulty: High

Recommended Action: Fix Design

Partially addressed by client

Description

As an extension of finding [A1](#), another missing validation on the amplification range regards the time interval that could be used to control modifications over the amplification factor.

In the current protocol state, there is no time/block range restriction for the change of the amplification factor. This may open the pool to [sandwich attacks](#), draining the pool's liquidity if the pool's `amplification` field is changed too greatly by `update_pool` function.

Recommendation

While it is comprehensible that the governance module is going to be responsible for validations on intervals of the amplification factor modifications, it is valid to evaluate the possibility of adding checks asserting that a certain amount of time has passed between the modification of the previous amplification factor and the updated one. Incrementally ramping the `amplification` will reduce the profit of a sandwich attack, potentially turning it into a financially unfeasible exploitation vector for the attacker.

Status

The client has partially addressed the finding by implementing a minimum window of blocks in which the modification of the amplification factor can happen. The progression of the amplification factor is gradual during this time window.

In theory, it is still possible to perform this exploit by using two blocks instead of one, as a single block is the minimum time window between the initial and final amplification factor progression. Still, the modification of the client hinders the attacker, as not only would they have to take over governance, but there would also be a one block window for arbitrageurs to interfere with the actions of any malicious users attempting to explore this vulnerability.



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 deviates 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

Severity: Informative

Recommended Action: Fix Code

Partially addressed by client

Description

Here are some 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 be followed.

- At location `math/src/stableswap/math.rs` line `170`, `reserve` variable holds value `xp[idx]`, however there is a reference to `xp[idx]` directly instead of `reserve`;
- Certain variables, such as the previously mentioned `xp`, do not have descriptive names. This affects the readability of the code;
- To a degree, the asset tradability state for buy and sell functions does not have much meaning if removing liquidity for the asset in question is still allowed. The user could bypass this by depositing an asset and withdrawing the different, desired asset.

Recommendation

- Replace `xp[idx]` reference to `reserve`;
- Where possible, rename variables with more descriptive names, avoiding acronyms. For instance, `xp` could be renamed to `filtered_reserves`;
- A more comprehensive way of controlling the tradability of assets must be developed. If purchases or sales of assets are disabled, then so should the addition and removal of liquidity be disabled accordingly.

Status

This finding has been partially addressed by the client, as the substitution of `xp[idx]` by `reserve` has been implemented.

[B2] Fees are unbounded

Severity: Informative

Recommended Action: Fix Code

Not addressed by client

Description

A created pool charges two fees, a **withdraw fee** and a **trade fee** whose values are independent, i.e. `withdraw_fee != trade_fee` is a valid state for the pool. The protocol places no restriction on the range of fees available, meaning the fees can range from 0 to 100 of the value of the withdrawal or trade. These fee values are determined solely by the governance protocol, and are able to be changed as frequently as governance allows.

To protect themselves from unfavorable transactions, users provide arguments to `min_buy_amount` and `max_sell_amount` parameters for `sell` and `buy` functions respectively. No such parameter exists for `remove_liquidity_one_asset`. Users should be warned that pools can validly exist with abnormally high fees, and spurious values for `min_buy_amount` and `max_sell_amount` or careless withdrawal of assets could incur unexpected costs.

The unbounded range on fees means that it is possible to set `withdraw_fee` low enough that it is cheaper for the user to perform a swap of tokens without using the `buy` and `sell` functions, but instead performing the swap with `add_liquidity` with the asset to sell followed by `remove_liquidity_one_asset` with the asset to buy. The lack of deviation from the curve implementation where fees are removed from `add_liquidity` means that careful consideration should be taken when choosing a value for `withdraw_fee` - it must be set high enough to ensure that `buy` and `sell` are more beneficial to the user for swaps, but also low enough to encourage liquidity providers.

Status

The client has acknowledged this finding.



[B3] `set_asset_tradable_state` does not check if the `asset_id` input is in the pool

Severity: Informative

Recommended Action: Fix Code

Addressed by client

Description

In HydraDX's stableswap pools, the administrator is capable of controlling the "tradability" state of all liquidity assets in the pool. It is possible to disable the capability of users to sell, buy, deposit, withdraw, or to completely freeze the assets in the pool by calling the `set_asset_tradable_state` function, informing the pool id (`pool_id`), asset id (`asset_id`), and asset state (`state`) to be modified.

Still, the `set_asset_tradable_state` function does not check if the parameter `asset_id` is in the parameter pool. The function relies on the governance authority to prevent setting the tradability of an asset that is not in the pool.

Recommendation

Add an assertion to check if the `asset_id` is in the pool.

Status

The client has addressed the finding by following the recommendation provided above.



[B4] The absence of a token in a pool may lead to misleading error messaging

Severity: Informative

Recommended Action: Fix Code

Addressed by client

Description

In several points of the code, the asset tradability state is checked in order to verify if a specific operation can be performed. For instance, if an asset is frozen in the pool, a user attempting to deposit this asset will fail to submit his transaction, receiving an error message specifying the reason for it.

What may lead to confusion among the users of HydraDX's stableswap pools is the fact that at different points of the code, the tradability state is verified before checking if the asset is in the pool. If the asset is in fact not in the pool, the user will retrieve an error message informing that the asset's state does not allow the operation to be performed, instead of informing that the asset isn't in the pool.

Recommendation

Whenever performing an operation, validate if the asset is in the pool before taking any action.

Status

The finding has been addressed by the modifications performed addressing finding [B3](#).

[B5] Presence of identified potential vulnerabilities in dependencies of crates

Severity: Informative

Not addressed by client

Description

By using the `cargo-audit` functionality, it is possible to match the analyzed code against the RUSTSEC security advisory database to identify possible exploitation vectors or code improvements related to dependencies of the code. Using this functionality in the audited repository, the following lists of potential vulnerabilities and warnings, alongside reference to more information, were found:

```
Crate:    time
Version:  0.1.45
Title:    Potential segfault in the time crate
Date:     2020-11-18
ID:       RUSTSEC-2020-0071
URL:      https://rustsec.org/advisories/RUSTSEC-2020-0071
Severity: 6.2 (medium)
Solution: Upgrade to >=0.2.23
```

```
Crate:    mach
Version:  0.3.2
Warning:  unmaintained
Title:    mach is unmaintained
Date:     2020-07-14
ID:       RUSTSEC-2020-0168
URL:      https://rustsec.org/advisories/RUSTSEC-2020-0168
```

```
Crate:    parity-wasm
Version:  0.45.0
Warning:  unmaintained
Title:    Crate `parity-wasm` deprecated by the author
Date:     2022-10-01
ID:       RUSTSEC-2022-0061
URL:      https://rustsec.org/advisories/RUSTSEC-2022-0061
```

```
Crate:    atty
Version:  0.2.14
Warning:  unsound
Title:    Potential unaligned read
Date:     2021-07-04
ID:       RUSTSEC-2021-0145
URL:      https://rustsec.org/advisories/RUSTSEC-2021-0145
```



All the links referenced above have been accessed up to June 23, 2023.

Recommendation

Ideally, the crates which are triggering the mentioned warnings could be either updated or replaced with adequate ones, depending on the scenario, risk, and available solution, but it is important to highlight that **the dependencies that are triggering such issues and warnings are from Polkadot's substrate itself**. In other words, **the solutions to the issues mentioned above are outside the HydraDX team's current scope, as they are part of the infrastructure required for constructing such pallets**.

Status

The client has acknowledged this finding.



[B6] Deviation from Curve stableswap invariant implementation and calculation strategy

Severity: Informative

Not addressed by client

Description

As mentioned in [Differences From Canonical Stableswap Invariant section](#), there is a deviation in the implementation of HydraDX from the original Curve finance implementation. We have not detected any vulnerabilities or attack vectors that the protocol can suffer from these changes, and all analysis and testing indicated that the difference in the values returned between the protocols maintains the health of the protocol. The original Curve stableswap invariant has been tried and tested for many years, with a tremendous amount of analysis performed to discover bugs and attack vectors. Any deviation from this should be done with extreme caution, and extensive analysis, as small edits can create subtle changes that expose new vulnerabilities, attack vectors, or undefined behavior that was not present in the previous protocol.

Furthermore, another difference in the mechanics of the protocol's calculations from Curve is the number of iterations performed using the Newton-Raphson's method (in the functions `calculate_d` and `calculate_y`). While Curve's original implementation uses up to 256 iterations in its calculations in order to search for a convergence point, HydraDX's stableswap protocol uses at most 64 iterations in the calculation of D and 128 iterations in the calculation of Y. The values generated in these calculations will not be rejected in case there is no convergence.

The current test suite available only covers 2 token pools, however it is dangerous to infer that properties checked holds for `2 < n` without explicit analysis or tests.



Recommendation

Perform a thorough analysis of changes to the protocol to demonstrate the changes to do not compromise the safety of the protocol. Extend all tests from only 2 token pools to 3,4,5 token pools.

Status

The client has acknowledged this finding.